COGNEX

CVL 9.0 Vision Tools Guide

2019 October 02
Revision: 9.0.0.1

Preface

This manual describes CVL vision tools and how you use them to solve vision applications. For specific programming information, refer to the programming reference documentation for your framework.

Style Conventions Used in This Manual

This manual uses the style conventions described in this section for text and software diagrams.

Text Style Conventions

This manual uses the following style conventions for text:

italic	Used for names of variables, data members, arguments, enumerations, constants, program names, file names. Used for names of books, chapters, and sections. Occasionally used for emphasis.
`courier`	Used for C/C++ code examples and for examples of program output.
`bold courier`	Used in illustrations of command sessions to show the commands that you would type.
<italic>	When enclosed in angle brackets, used to indicate keyboard keys such as *<Tab>* or *<Enter>*.

Microsoft Windows Support

Cognex CVL software runs on Windows operating systems.

In this documentation set, these are abbreviated to Windows unless there is a feature specific to one of the variants. Consult the *Getting Started* manual for your CVL release for details on the operating systems, hardware, and software supported by that release.

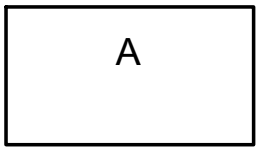
Software Diagramming Conventions

This manual uses the following symbols in class diagrams:

Classes are shown as a box with the class name centered inside the box. For example, a class A with the C++ declaration

```
class A{};
```

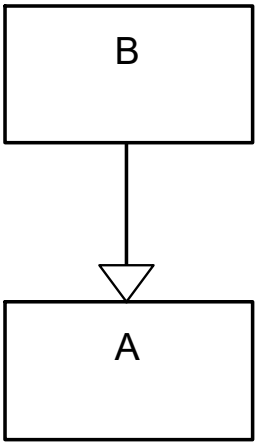
is shown graphically as follows:



Inheritance relationships between classes are shown using solid-line arrows from the derived class to the base class with a large, hollow triangle pointing toward the base class. For example, a class B that inherits from a class A with the declaration

```
class B : public A {};
```

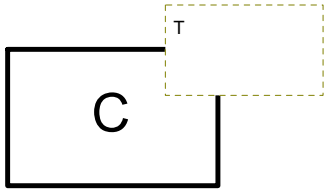
is shown graphically as follows:



Template classes are shown as a class box with a smaller, dotted-line rectangle representing the template parameter superimposed on the upper right corner of the class box. For example, a template class C with a parameter of type class T with the declaration:

```
template                <class T>
class C{};
```

is shown graphically as follows:



These symbols are based on the Unified Modeling Language (UML), a standard graphical notation for object-oriented analysis and design. See the latest *OMG Unified Modeling Language Specification* (available from the Object Management Group at http://www.omg.org) for more information.

Cognex Offices

The following are the address and phone number of Cognex Corporate Headquarters, and the address of the Cognex web site:

Corporate Headquarters	Cognex Corporation Corporate Headquarters One Vision Drive Natick, MA 01760-2059 (508) 650-3000
Web Site	www.cognex.com

Table of Contents

What's New in This Release

The following table summarizes the information added or changed to the base release of this manual to create this release. The manual might also contain smaller editorial changes and enhancements (including updates in formatting) not listed in this table.

Note: This section is for Cognex internal documentation review purposes only, it will not be present in the published manual.

Location	Description	Feature
Cognex CVL software runs on Windows operating systems. on page 2	OS updates	
Some Useful Definitions on page 158, *Setting the Origin* on page 167	Adding Data Matrix fiducial checkerboard calibration plate support	

Note: An N/A in the Feature column above indicates that the update described was made to conform to a change request not related to a specific feature.

Featurelets

This chapter describes Featurelets, a mechanism used by vision tools to describe points along boundary of features in images or points along the boundary of shape models. A featurelet is described by a location, an angle, a magnitude, a weight, and an orientation modulus.

This chapter has the following sections:

Featurelets Description on page 19 describes what featurelets are and how vision tools use them.

Using Featurelets on page 21 describes how you use featurelets with vision tools and your vision application.

Featurelets Description

A featurelet is represented by a C++ class (**ccFeaturelet**) that provides a simple way to describe a point in an image. Featurelets have the following characteristics:

position The featurelet x,y location. All featurelets have a position.

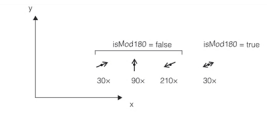
angle The featurelet orientation. All featurelets have an angle.

weight An importance factor you can assign to a featurelet. Weight is optional and has a default value of 1.

magnitude A size factor you can assign to a featurelet. Magnitude is optional and has a default value of 0.

isMod180 A boolean defining if the featurelet is unique through 360° or only 180°. For modulus 180 featurelets, there is no difference between the featurelet at 0° or at 180°.

Featurelets are typically shown as a dot indicating the position, with an overlaid arrow to indicate the featurelet orientation. See the examples in the figure below.

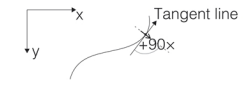


Featurelet example

The *position* is the point in x,y space where the featurelet is located. The featurelet angle is shown by the orientation of the arrow. A featurelet with an arrowhead on both ends indicates that isMod180 = true. This means that if you rotate the featurelet 180° its description does not change.

Featurelet Chains and Chain Sets

Featurelets describe boundaries or contours of shapes or features in images. Featurelet *positions* are points along a boundary path, and the default convention is for featurelet *angles* to be +90° from the tangent direction at each point. It is assumed that featurelets point in the direction of the gradient, from dark to light. See the figure below.



Featurelet angle

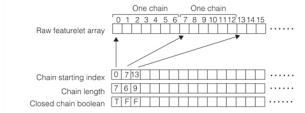
When describing boundaries and contours it is convenient to group all of the featurelets along one continuous boundary into a chain. The **ccFeatureletChainSet** class does this and encapsulates a number of chains (a set of chains). There is no class for a single chain. A single chain is represented by a **ccFeatureletChainSet** object where the number of encapsulated chains is one.

The figure below is an example of a featurelet chain.

Featurelet chain example

A featurelet chain can be either open or closed and can contain any number of featurelets, including zero. The figure above is an example of an open chain. A circle is an example of a shape that can be represented by one closed chain.

The **ccFeatureletChainSet** class includes a vector of raw featurelets. Each chain in the set corresponds to a contiguous subset of these raw featurelets. The featurelet chain set also includes a vector of chain start indices and a vector of chain lengths. These vectors combine to specify the contiguous featurelet subsets corresponding to each chain. The featurelet chain set also includes a vector of booleans corresponding to whether each chain is open or closed. See the figure below.



ccFeatureletChainSet internal arrays

Note that every featurelet in the raw vector is an element of at most one chain; that is, there are no shared featurelets in the raw vector.

However, it is legal for a featurelet to belong to no chain. The most typical use cases are the following:

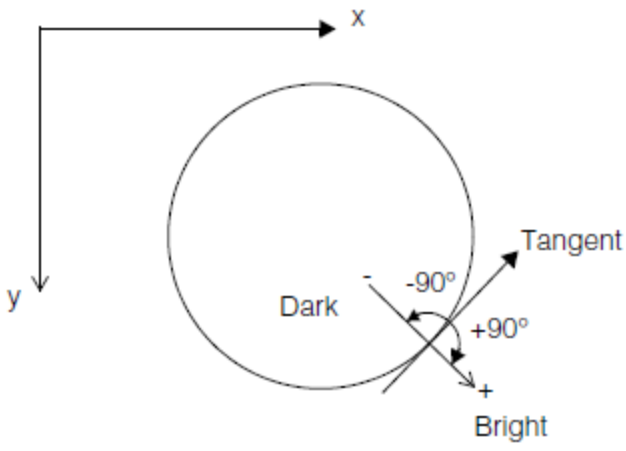
1. A set of featurelets where every featurelet belongs to exactly one chain, that is, there are no unchained featurelets.
2. A set of featurelets with zero chains, that is, no featurelets are chained.

However, in the most general case, some featurelets might belong to chains while other featurelets might not belong to any chain. Users who wish to iterate through all featurelets regardless of whether or not those featurelets are chained should use the featurelets() vector getter; users who wish to iterate through only chained featurelets should use the featureletChains() vector getter.

Each featurelet chain is composed of a contiguous set of featurelets in the raw featurelet array.

Featurelet Chain Polarity

The CVL shapes convention is that default polarity corresponds to bright in the direction of increasing angle with respect to the tangent direction along the shape. This gradient vector points from dark to bright and represents positive polarity. See the figure below.



CVL shapes default polarity

The featurelet chain polarity follows this convention (assuming the featurelet angle corresponds to the gradient angle). The featurelet orientation is in the direction from negative to positive (dark to bright).

The polarity of a featurelet chain is defined by the angle and position of its individual featurelets. To determine the chain polarity from its individual featurelets, we provide the **ccFeatureletChainSet::proportionPositiveCrossProduct()** member function. This function analyzes the specified chain and returns a number in the range 0 - 1.0 indicating what percentage of the chain featurelets have a positive cross product between both incident vectors (between successive featurelet positions) and the angle which is assumed to be the gradient angle. If it returns a value less than 0.5 the chain has a negative polarity. If it returns a value greater than 0.5 the chain has positive polarity. Note that featurelets with *isMod180* = true are not included in the computation. You should be suspect if the function returns exactly 0.5. This likely means that all the featurelets are *isMod180* or that the chain contains no featurelets.

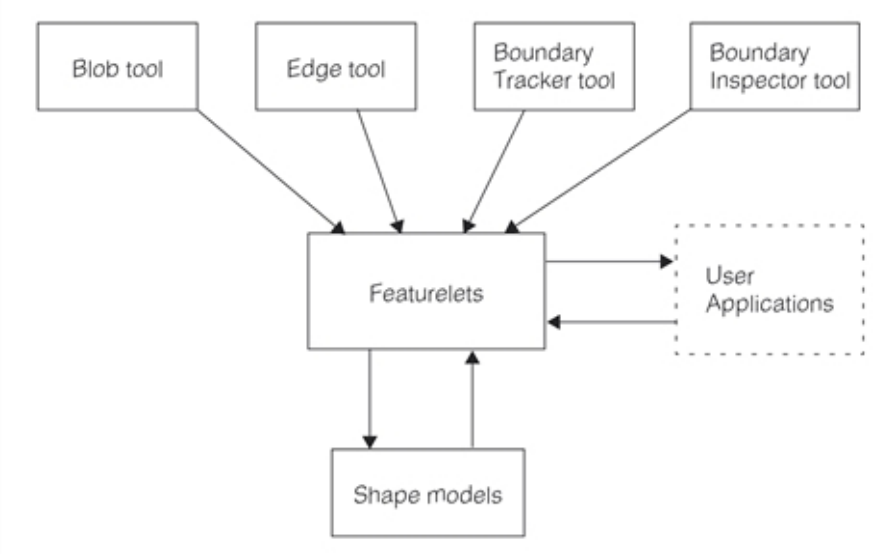
For closed chains the combination of **ccFeatureletChainSet::isRightHanded()** and **ccFeatureletChainSet::proportionPositiveCrossProduct()** define the *inside polarity* of the shape described by the chain; that is, the polarity of the shape interior compared to the shape exterior. See the table below.

isRightHanded()	proportionPositive CrossProduct()	Inside Polarity
true	< 0.5	Dark on light (negative)
	> 0.5	Light on dark (positive)
false	< 0.5	Light on dark (positive)
	> 0.5	Dark on light (negative)

Determining inside polarity

Using Featurelets

Featurelets and featurelet chain sets provide a way for Cognex vision tools to share a common format for vision tool inputs and outputs. This is a recent effort to make it easier for one application to process results from various vision tools. For example, an application you write might analyze featurelet chain sets that are produced by the Blob tool or the Edge tool. Vision tools that currently support featurelets are shown in the figure below.



Featurelet vision tool support

The Boundary Inspector tool was designed to use featurelets while the Blob tool, Edge tool, and the Boundary Tracker tool all contain global functions that allow you to convert their tool results to featurelet chain sets. In addition, shape models can be converted to featurelet chain sets and featurelet chain sets can be converted to a

geometric statistics using the shape model functions.

Converting to Featurelets

The conversion of vision tool results and shape models into featurelets is somewhat different in each case. The tables below summarizes how the conversion is done for each of the featurelet *position*, *angle*, *weight*, *magnitude*, and *isMod180* parameters.

Position

Source	Featurelet position
Edge tool	The edgelet *positions* mapped by the edgelet set's *clientFromImageXform.*
Blob tool	Same as positions of *boundaryChainCode* elements, transformed by the blob scene's *clientFromImageXform*.
Boundary Tracker tool	The boundary tracker points in client coordinates.
Shape model	The sample positions for the shape model.

Angle

Source	Featurelet angle
Edge tool	Compute the edgelet tangent angle by subtracting 90° from the edgelet's **angle()** and then mapping that tangent angle by the edgelet set's *clientFromImageXform*. The featurelet angle is then computed as the normal to this mapped tangent angle.
Blob tool	The tangent angle at each point is computed in image coordinates as the average of the two incident vectors to the two adjacent points, and in the direction of the walk in the image. This tangent angle is then mapped by the scene's *clientFromImageXform*. The featurelet angle is computed as the normal of this mapped tangent angle, in client coordinates.
Boundary Tracker tool	*angle* normal to the boundary tracker angle and in the direction towards the white side of the shape. The featurelet angles are in the gradient direction (from negative to positive) normal to the **ccBoundaryTrackerPoint** angles which are in the tangent direction.
Shape model	If the sampling parameter **computeTangents()** is true, define the *angle* for each featurelet as the normal to the tangent angle of the corresponding sample point on the positive side of the shape. The positive side is determined by the effective polarity of that portion of the shape from which the feature was sampled. If **computeTangents()** is false, the featurelet angles are undefined.

Weight

Source	Featurelet weight
Edge tool	Set to 1.
Blob tool	Set to 1.
Boundary Tracker tool	Set to 1.
Shape model	The weight of a featurelet is the product of the effective weight along that portion of the shape from which the feature was sampled and the weight scale supplied by *params*.

Magnitude

Source	Featurelet magnitude
Edge tool	Same as edgelet magnitude.
Blob tool	Set to 0.
Boundary Tracker tool	Set to 0.
Shape model	The magnitude of a featurelet is the product of the effective magnitude along that portion of the shape from which the feature was sampled and the magnitude scale supplied by *params*.

isMod180

Source	Featurelet isMod180
Edge tool	Set to false.
Blob tool	Set to false.
Boundary Tracker tool	Set to false.
Shape model	Set the *isMod180* flag for each featurelet to true if and only if the effective ignore polarity flag for that portion of the shape from which the feature was sampled is true.

Featurelet Filters

After using a vision tool to find all of the featurelets in an image you may find you have many featurelets that are not pertinent to your application. To improve your application efficiency you may wish to eliminate featurelets you do not wish to process. You can do this with a featurelet filter.

Cognex provides a family of filter classes you can use to filter vectors of featurelets and vectors of featurelets. These classes are described below:

ccFeatureletFilter
This is the abstract base class for featurelet filters and featurelet chain filters. All of the Cognex filters described below derive from this base class. If you decide to write your own featurelet filter you must also derive from **ccFeatureletFilter**. Information on how to create your own filter class is covered in the **ccFeatureletFilter** reference page.

The **ccFeatureletFilter** base class contains the function **filterFeaturelets()** that you call to filter vectors of featurelets, and the function **filterChains()** you call to filter vectors of featurelet chain sets. The base class also contains the **isInverted()** function you can call to invert the filter. If you derive your own filter you must override the protected function **filter_()**.

Note that all Cognex filters support featurelet chain filtering however, some filters do not support featurelet filtering. Cognex filters are described in the following paragraphs.

ccFeatureletFilterComposite
To use this filter you provide a vector of filters that are then run in sequence where each filter result is the input to the next filter. For example, if you call **filterFeaturelets()**, the function is run on all the filters in the filter vector, in vector order. The filtering performed is equivalent to, but more efficient than, calling **filterFeaturelets()** of each of the filters individually. Using **ccFeatureletFilterComposite** and the **isInverted()** base class function you can form filters capable of complex filtering operations such as logical and, and exclusive or operations.

ccFeatureletFilterBoundary
This filter provides functionality for filtering featurelets based on their positions, orientations, and polarities compared to a given **ccShape** boundary. You specify a distance tolerance, an angular tolerance, and whether to consider polarity.

Using this filter class you can call **filterFeaturelets()** to filter featurelets, or **filterChains()** to filter featurelet chain sets. In both cases each featurelet is analyzed as follows:

- Each featurelet is compared with sampled points on the model boundary that are within the distance tolerance. If there are no such points, the featurelet survives and is not filtered out.
- For points within the distance tolerance, the angular differences between the tangents of these points and the featurelet tangent angle are measured. If any of these angular differences is less than, or equal to the angular tolerance, the featurelet is filtered out. Otherwise, the featurelet survives. If you choose to consider polarity, it may have an impact on the measured angular difference. See the **ccFeatureletFilterBoundary** reference page for more details.

ccFeatureletFilterMagnitudeHysteresis
This filter class provides functionality for filtering featurelet chains based on hysteresis magnitude of the featurelets and their chains. You call **filterChains()** to filter featurelet chain sets. Calling **filterFeaturelets()** to filter featurelets is not supported by this filter.

This class allows you to filter featurelets based on a featurelet magnitude range. The magnitude is a featurelet property that represents its size.

ccFeatureletFilterLength
This filter class provides functionality for filtering featurelet chains based on their length, regardless of their open or closed status. You call **filterChains()** to filter featurelet chain sets. Calling **filterFeaturelets()** to filter featurelets is not supported by this filter.

ccFeatureletFilterRegion
This class filters featurelets based on whether they are inside or outside of a geometric region that you specify. The class can filter both featurelets and featurelet chain sets.

Each filter retains its run-time behavior as described in the preceding sections.

The Cognex filter classes are summarized in the table below.

	Featurelets	Featurelet chains	Filter criteria
ccFeatureletFilterBoundary	Yes	Yes	distance, angle, polarity
ccFeatureletFilterMagnitudeHysteresis	No	Yes	featurelet magnitude
ccFeatureletFilterLength	No	Yes	chain length
ccFeatureletFilterRegion	Yes	Yes	featurelet location
ccFeatureletFilterComposite	Any sequence of the above filters. See **ccFeatureletFilterComposite** *ccFeatureletFilterComposite To use this filter you provide a vector of filters that are then run in sequence where each filter result is the input to the next filter. For example, if you call filterFeaturelets(), the function is run on all the filters in the filter vector, in vector order. The filtering performed is equivalent to, but more efficient than, calling filterFeaturelets() of each of the filters individually. Using ccFeatureletFilterComposite and the isInverted() base class function you can form filters capable of complex filtering operations such as logical and, and exclusive or operations.* on page 23		

Filter class summary

Shape Models

This chapter describes the design and use of shape models in CVL. It contains the following sections:

The first two sections, this one and *Some Useful Definitions on page 25*, provide an overview of the chapter and define some terms that you will encounter as you read.

Shape Model Overview on page 25 provides an overview of shape models and their use with PatMax.

Shape Model Properties on page 26 describes the properties that shape models add to shapes.

Using Shape Models on page 37 explains how you use shape models with vision tools.

Model Maker on page 37 provides an overview of some tools that you can use to extract shapes and shape models from images.

Some Useful Definitions

magnitude: A shape model property that specifies the strength of a boundary.

polarity: A shape model property that specifies the gradient direction for a boundary. The +90° side of a boundary is *positive* (brighter) and the -90° side is *negative* (darker) by default. If polarity is reversed, the opposite is true.

pure shape: A shape derived from **ccShape** with no explicit shape model properties (polarity, weight, or magnitude information).

shape description: A geometric shape, described from a class derived from **ccShape**, representing the high-contrast boundaries of an object in an image. A shape description represents shape model properties (polarity, weight, and magnitude information) for each boundary it defines. These properties can be implicit (represented by a pure shape) or explicit (represented by a shape model). PatMax can be trained using shape descriptions. Also called a *geometric description*.

shape model: A shape derived from **ccShape** with explicit shape model properties (polarity, weight, and magnitude information).

weight: A shape model property that specifies how important a shape is in relation to other shapes. PatMax uses this information, for example, when calculating scores for pattern matching.

Shape Model Overview

Shape models provide the ability to assign properties to shapes used to model the appearance of objects in an image. These shape modeling properties are required for certain vision tools, such as PatMax.

Geometrically modeling the appearance of an object in an image involves describing the visible features of the object using geometric shapes. Visible features of an object include boundaries, contours, and regions that the shape encloses. Modeling the appearance of an object also entails specifying certain properties that cannot be described by pure geometry alone. Such properties include appearance attributes that can be determined by both object characteristics, such as variability of physical attributes, and imaging conditions, such as lighting and optics.

CVL currently supports the following shape model properties:

- **Weight:** specifies the relative importance of a shape in comparison to other shapes. Shapes with higher weight contribute more to an overall score calculation than those with lower weight. The default weight is 1.0.
- **Polarity:** specifies the gradient direction for a boundary. The +90° side of a boundary is *positive* (brighter) and the -90° side is *negative* (darker) by default. The *polarity reversal flag* specifies whether the normal polarity of a shape is reversed. The *ignore polarity flag* specifies whether polarity information should be ignored for a shape. The default for both flags is *false*.

- **Magnitude:** specifies the strength of a boundary as a non-negative value. The magnitude property can be used, for example, to denote the edge contrast of an object modeled by a shape. The default magnitude is 1.0.

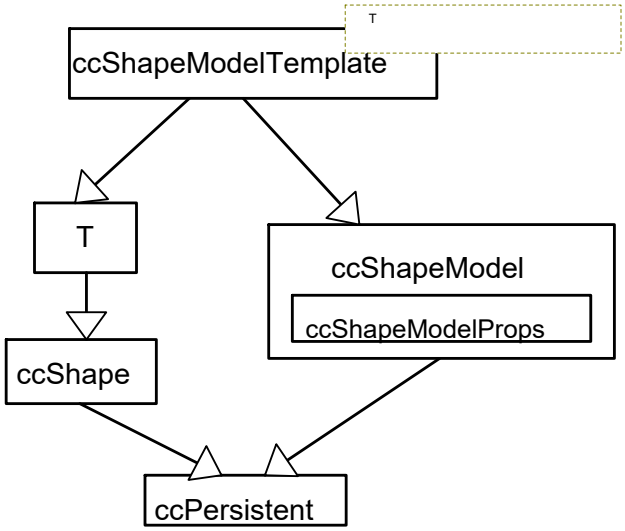
You can specify model properties for primitive shapes, entire shape trees, or even internal nodes of a shape tree. Properties applied to a shape tree implicitly affect all of its descendents. The effective properties for a leaf node of a tree are, therefore, a combination of the properties specified by all of its ancestors.

Note: The boundary shape of a region tree is considered a descendant of the region tree for the purpose of calculating the effective properties of a node. See *Setting ccRegionTree Model Properties* on page 34 for more information.

It is only necessary to specify shape model properties when specifically warranted by a particular vision tool or application for which they are being used. For example, if you are using a shape with PatMax for alignment and polarity is to be ignored (that is, **ignorePolarity()** == true), then it is not necessary to specify polarity for that shape.

See the *Shapes* chapter of the *CVL User's Guide* for more information on shapes and shape trees.

The **ccShapeModel** class adds explicit weight, polarity, and magnitude information to **ccShape**-derived shapes. This information is added to the shape by deriving a **ccShapeModelTemplate<>** object, instantiated with a particular shape, from both that shape and the **ccShapeModel** class. The resulting shape model object inherits shape information from the shape class that the template was instantiated with, and weight, polarity, and magnitude information from the **ccShapeModel** class.



ccShapeModelTemplate<> class inheritance hierarchy

See *Using Shape Models* on page 37 for an explanation of how the **ccShapeModelTemplate<>** class is instantiated with various shape types.

Note: It is currently not possible to instantiate your own shape models by instantiating your own templates. The only supported shape model types are those that Cognex provides. See *Using Shape Models* on page 37 for a list of the supported shape model types.

Shape Model Properties

Shape models impart explicit shape model properties to shapes, including weight, polarity, and magnitude. For pure shapes, these properties are implicit.

Weight

The *weight* property determines the relative importance of a shape. In the context of a shape tree, the weight of a child determines the relative importance of the features modeled by that child in relation to the overall object modeled by the shape tree. This assignment of relative importance is often dependent on an application. For example, when using a shape model to perform an alignment (for example, using PatMax), the weight determines how much a particular shape will contribute to the alignment score.

The assigned weight for a shape can be zero, positive, or negative. In general, the larger the absolute value of the weight, the higher its relative importance for the application:

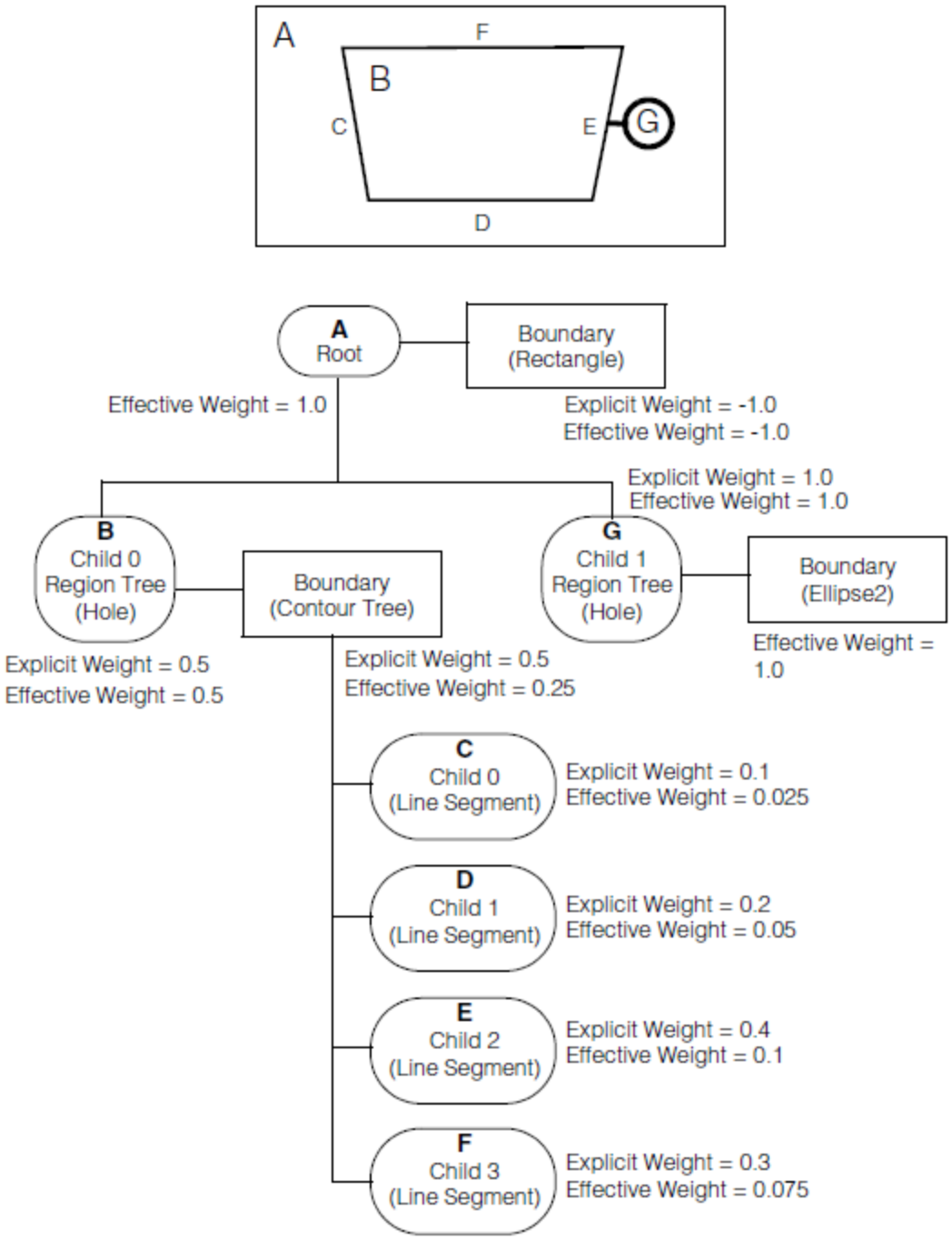
- A *high positive* weight means that the features modeled by the shape are very important, are accurate, and are consistently visible in the image.
- A *low positive*, or *zero*, weight means that the features modeled by the shape are relatively unimportant, inaccurate, or inconsistently visible in the image.
- A *negative* weight means that these features should never be present in the image. For example, if a shape is being used for alignment, a negative weight would count negatively toward the alignment score (that is, the shape is expected to be absent).

For pure shapes (**ccShape**-derived objects that are not shape models), the weight property is implicitly set to 1.0 by default.

Effective Weight

The *effective weight* for a shape is 1.0 by default, if not explicitly specified for the shape or any of its ancestors. Otherwise, the effective weight is determined by multiplying any explicitly specified weights for the shape and all of its ancestors. If more than one explicitly specified weight for the shape and its ancestors is negative, the effective weight is 0.

The figure below shows an example of the effective weights of the individual nodes in a region tree shape description. Shapes with explicit weight designations are shape models (of type **ccShapeModel**). Those without explicit weight designations are pure shapes (of type **ccShape**, but not of type **ccShapeModel**), whose implicit weight is always 1.0.



Effective weights of individual nodes in a region tree shape description

Effective polarities of individual nodes in a region tree shape description on page 32 shows a complex region tree with multiple children with varying weights. The effective weights of the nodes in this region tree are as follows:

Node	Weight	*	Effective Weight of Immediate Ancestor	=	Effective Weight
A	1.0			=	1.0
A's boundary (rectangle)	-1.0	*	1.0	=	-1.0
B	0.5	*	1.0	=	0.5

Node	Weight	*	Effective Weight of Immediate Ancestor	=	Effective Weight
B's boundary (hole)	0.5	*	0.5	=	0.25
C	0.1	*	0.25	=	0.025
D	0.2	*	0.25	=	0.05
E	0.4	*	0.25	=	0.1
F	0.3	*	0.25	=	0.075
G	1.0	*	1.0	=	1.0
G's boundary (ellipse)	1.0	*	1.0	=	1.0

The effective weights of the nodes in this shape tree are as follows:

- Node A (the root) has an implicit weight of 1.0 and an *effective weight* of 1.0.
- Node A's boundary has an *effective weight* of -1.0, obtained by multiplying its explicit weight (-1.0) by the *effective weight* of its ancestor (A = 1.0). A negative *effective weight* means that this shape should not be present in the image.
- Node B has an *effective weight* of 0.5, obtained by multiplying its explicit weight (0.5) by the *effective weight* of its ancestor (A = 1.0).
- Node B's boundary has an *effective weight* of 0.25, obtained by multiplying its explicit weight (0.5) by the *effective weight* of its immediate ancestor (B = 0.5).
- Node C has an *effective weight* of 0.025, obtained by multiplying its explicit weight (0.1) by the *effective weight* of its immediate ancestor (B's boundary = 0.25).
- Node D has an *effective weight* of 0.05, obtained by multiplying its explicit weight (0.2) by the *effective weight* of its immediate ancestor (B's boundary = 0.25).
- Node E has an *effective weight* of 0.1, obtained by multiplying its explicit weight (0.4) by the *effective weight* of its immediate ancestor (B's boundary = 0.25).
- Node F has an *effective weight* of 0.075, obtained by multiplying its explicit weight (0.3) by the *effective weight* of its immediate ancestor (B's boundary = 0.25).
- Node G has an *effective weight* of 1.0, obtained by multiplying its explicit weight (1.0) by the *effective weight* of its ancestor (A = 1.0).
- Node G's boundary also has an *effective weight* of 1.0, obtained by multiplying its implicit weight (1.0) by the *effective weight* of its immediate ancestor (G = 1.0).

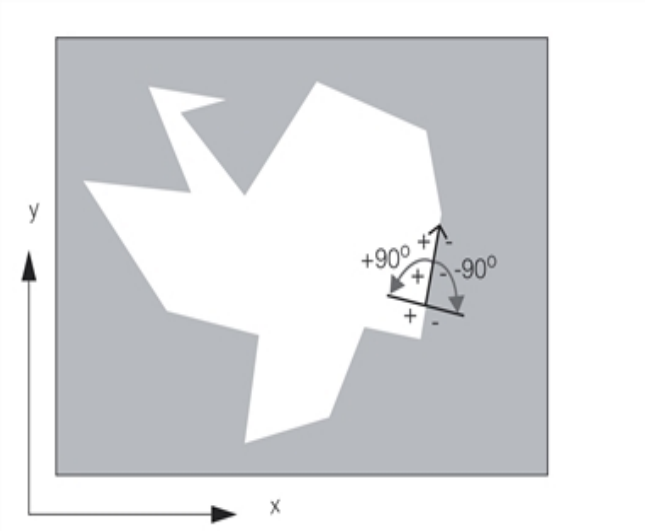
When you supply shape descriptions with the above weights to a tool such as PatMax, the shape modeled by node G (an ellipse) will contribute the most points (per unit length) to the score that the tool calculates for the run-time image. As points accumulate per unit length along the perimeter of the shape, a large shape with low weight may still contribute more points to the overall score than a small shape with high weight.

Polarity

The *polarity* property specifies the direction of the gradient normal to the boundary of the features being modeled. That is, polarity specifies which side of a boundary is *positive* and which side is *negative*.

When a shape is used, for example, to model the intensity boundaries (edges) of an object in an image, the polarity specifies the intensity gradient. In other words, it specifies which side of the boundary is relatively dark with respect to the other. By convention, the bright side of a boundary is positive and the dark side is negative.

Each shape has an unique *effective polarity*. The side of a shape boundary that is in the $+90^{\circ}$ direction from the tangent direction has one polarity, either *positive* or *negative*, and the side in the -90° direction from the tangent direction has the opposite polarity. By default, the $+90^{\circ}$ side of a shape boundary is *positive*, and the -90° side is *negative*.



Right-handed closed polyline with default polarity

When a shape is displayed in a left-handed coordinate system (for example, in image coordinates), the +90° and -90° directions correspond to clockwise and counterclockwise rotations of the unit tangent vector, respectively. In a right-handed coordinate system, the +90° and -90° directions are reversed.

When mapping a shape from one coordinate system to another with the opposite handedness (for example, using a transform with a handedness flip), you must reverse the polarity of the mapped shape to preserve its original polarity. You can accomplish this by toggling the *polarity reversal flag*.

Polarity Reversal

The *polarity reversal flag* determines whether the normal polarity for a shape is reversed. In other words, you can explicitly reverse the polarity of a shape by setting its polarity reversal flag. For pure shapes (**ccShape**-derived objects that are not shape models), the polarity reversal flag is implicitly set to false (not reversed) by default.

Setting Polarity

Right-handed closed shapes are positive on the inside and negative on the outside by default. Left-handed closed shapes have the opposite default polarities. Primitive shape regions with fixed handedness, such as circles and rectangles, are always positive on the inside and negative on the outside by default (that is, when they are not part of a region tree). The polarity of closed shapes for which handedness can vary (for example, closed contour trees, affine rectangles, polylines, generalized polygons, and region trees) is somewhat more complicated.

The **ccShapeModel** interface in CVL allows you to set and query the polarity of any region without respect to handedness, provided that the region does not self-intersect.

The **ccShapeModel::setInsidePolarity()** static function sets the polarities of the insides of region shapes. It does not affect non-region shapes. The default behavior of this function is to set the insides of all regions in a shape tree to positive. For example, to set the polarity of the insides of all children of a region tree model named *shape* to positive, you can call:

```
ccShapeModel::setInsidePolarity(shape);
```

You can set the polarity of the insides of region children to negative by passing a second argument of *false* to this function as follows:

```
ccShapeModel::setInsidePolarity(shape, false);
```

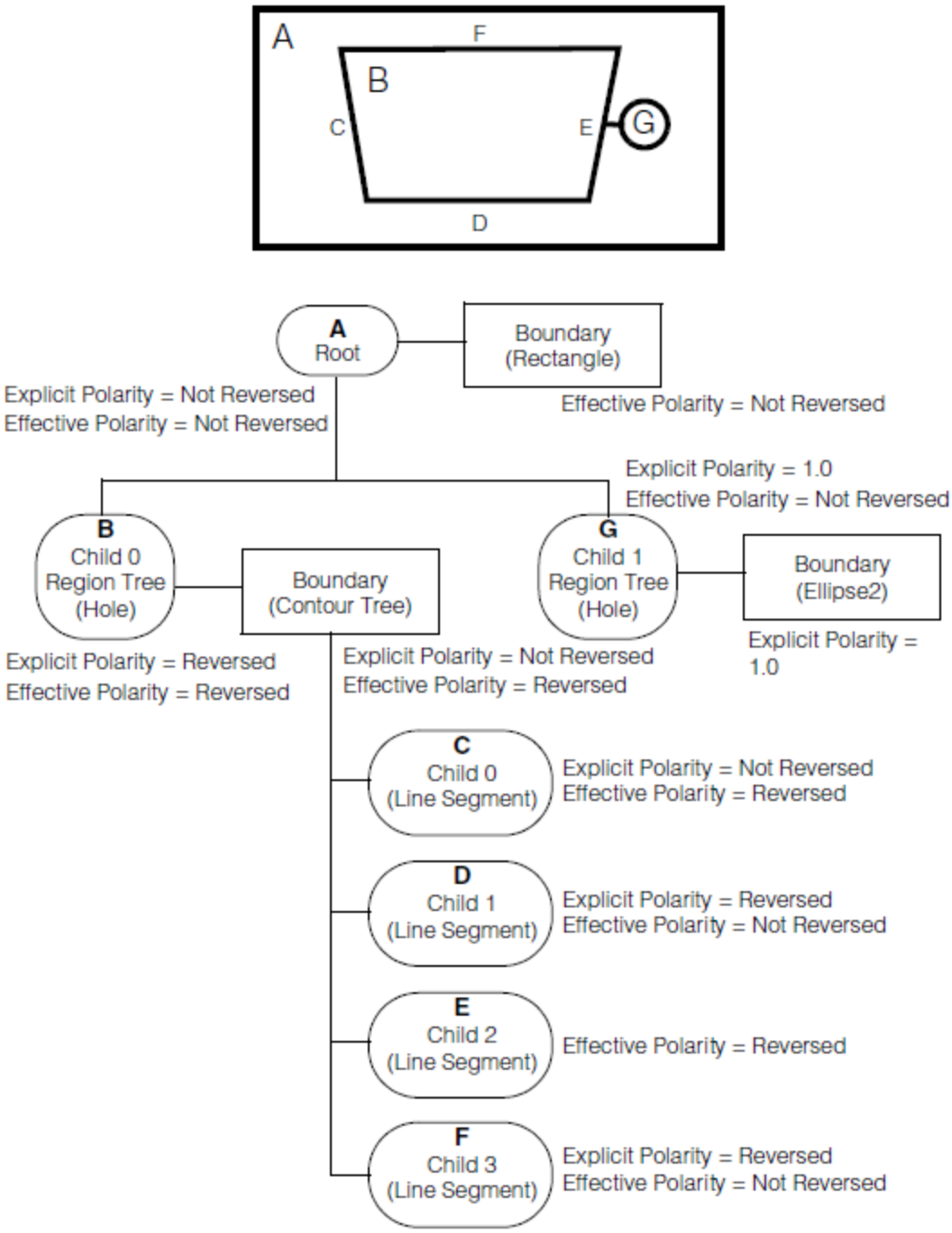
Querying Polarity

The **ccShapeModel::isInsidePolarity()** static function can be used to query whether all of the closed regions within a given shape have a consistent (positive or negative) polarity. This function ignores non-region shapes.

Effective Polarity

The *effective polarity* of a shape is determined by applying the explicitly specified polarity reversals for the shape and all of its ancestors. An even number of reversals results in no effective reversal, while an odd number results in an effective reversal.

Effective polarities of individual nodes in a region tree shape description on page 32 shows an example of the effective polarities of the individual nodes in a region tree shape description. Shapes with explicit polarity designations are shape models (of type **ccShapeModel**). Those without explicit polarity designations are pure shapes (of type **ccShape**, but not of type **ccShapeModel**), whose implicit polarity is always *not reversed*.



Effective polarities of individual nodes in a region tree shape description

The figure above shows a complex region tree with multiple children with varying polarities. The effective polarities of the nodes in this region tree are as follows:

Node	Number of Reversals (Including Ancestors)	Effective Polarity
A	0	Not reversed
A's boundary (rectangle)	0	Not reversed
B	1 (odd)	Reversed

Node	Number of Reversals (Including Ancestors)	Effective Polarity
B's boundary (hole)	1 (odd)	Reversed
C	1 (odd)	Reversed
D	2 (even)	Not reversed
E	1 (odd)	Reversed
F	2 (even)	Not reversed
G	0	Not reversed
G's boundary (ellipse)	0	Not reversed

The effective polarities of the nodes in this shape tree are as follows:

- Node A's boundary has an effective polarity of *not reversed*, because its implicit polarity is *not reversed*, and its ancestor, A, does not override that setting.
- Node B's boundary and node C have effective polarities of *reversed*, even though their explicitly assigned polarities are *not reversed*.
- Nodes D and F have effective polarities of *not reversed*, even though their explicitly assigned polarities are *reversed*.
- Node E has an effective polarity of *reversed*. It has no explicit polarity assigned. It has an implicit polarity setting of *not reversed*, which its ancestor, B, overrides.

Ignore Polarity Flag

The *ignore polarity flag* specifies whether vision tools should ignore polarity information for a shape. You can use this flag, for example, to effectively invalidate polarity information when a shape's polarity is unknown, or simply to ignore polarity information in cases where it is known.

The ignore polarity flag is *false* by default, which means that polarity information is valid for all shapes and that vision tools should use it when appropriate. You must explicitly set this flag to *true* to invalidate polarity information for a shape.

Effective Ignore Polarity Flag

The *effective ignore polarity flag* for a given shape is determined by logically ORing all of the ignore polarity flags explicitly specified for the shape and all of its ancestors. Setting the ignore polarity flag to true for any shape effectively invalidates polarity information for the shape and all of its descendants. Thus, you can tell vision tools to ignore polarity for an entire subtree of a shape tree by setting the ignore polarity flag for the root of the subtree to true.

Note: Some modes of Cognex vision tools do not support the ignore polarity flag. Such limitations are documented in the public header files for these tools.

Magnitude

The *magnitude* property specifies the non-negative strength of the boundary modeled by a shape. The magnitude can, for example, denote the edge contrast of the object that the shape is modeling. Each vision tool may interpret magnitude differently. See the header file for the respective tool to determine how that tool interprets the magnitude property for a shape.

The magnitude of all shapes is 1.0 by default. You can explicitly set the magnitude property to any non-negative value as desired.

Effective Magnitude

The *effective magnitude* of a shape is determined by multiplying the explicitly specified magnitudes for the shape and all of its ancestors.

Setting Shape Model Properties

You can specify the polarity and weight of a shape as follows:

1. To explicitly specify a polarity of *reversed*, pass a value of *true* as the first argument to the constructor for a **ccShapeModelProps** object. If you do not specify a value, the polarity will be *not reversed* by default.

   To specify a weight, pass a value between -1.0 and +1.0 to the **ccShapeModelProps** constructor.

   For example, the following code constructs a shape model properties object with a polarity of *reversed* and a weight of 0.9:

   ```
   ccShapeModelProps modelProps(true, 0.9);
   ```

2. To add these properties to a shape, construct the appropriate type of shape model (see *Using Shape Models* on page 37), passing the **ccShapeModelProps** object as an argument. For example, to create a rectangle shape model with the above properties, you could use:

   ```
   ccRectModel rectModel(ccRect(cc2Vect(0,0), cc2Vect(100,100)),
    modelProps);
   ```

Setting ccContourTree Model Properties

In general, the shapes within a contour tree (whether open or closed) should have consistent polarities, weights, magnitudes, and ignore polarity flags. This is typically accomplished by setting model properties for the contour tree, which automatically sets the properties for all children. Consistency of model properties is present by default. However, it is possible to set different polarities for individual children of a contour tree.

Setting ccRegionTree Model Properties

Similarly, the boundary shapes of a region tree should, in general, have polarities that are consistent with their status (whether they are hole or solid regions) within the tree. The *inside* of the region tree should be of one polarity, and the *outside* should be of the opposite polarity. Furthermore, the effective weights of shapes within a region tree should all be of the same sign. This consistency is also present by default, such that the inside of the region tree is *positive* and the outside is *negative*. However, such consistency is not a requirement, and it is possible to set the model properties independently for each boundary shape within the region tree, regardless of hole status.

Note: Vision tools generally ignore any *phantom* hole regions at the root of a region tree, disregarding their effective model properties.

It is important to keep in mind that the effective handedness, and therefore tangent direction and default polarity, for a region tree boundary shape is governed by whether the shape is a hole or not. Therefore, assigning a particular region shape as the boundary of a region tree may actually have the effect of reversing its effective polarity within the tree, regardless of the polarity of the boundary shape itself.

Note: Any new line segments arising from clipping the boundary of a region tree will not have any explicit model properties assigned and will therefore have their effective model properties determined by the effective model properties of the region tree itself.

For example, if the polarities of the individual children of a contour tree are all explicitly reversed, and the contour tree is then used to specify the boundary of a region tree that is itself not reversed in polarity. If that contour tree boundary is clipped, then any resulting new line segments included as part of the new contour tree boundary will not be effectively reversed, even though the remaining original segments of the new contour tree will be effectively reversed (as they were before the clipping).

Setting ccFLine Model Properties

The **ccFLine** shape does not have a well-defined tangent direction and, therefore, does not have well-defined polarity. For this reason, Cognex recommends using **ccLine** when possible instead of **ccFLine**.

Setting ccGenPoly and cc2Wireframe Model Properties

Shapes of type **cc2Wireframe** are special in that they already contain shape model information in addition to the pure geometry provided by the **ccGenPoly** base class. In particular, they already contain polarity information.

Note: The effect of reversing the polarity of a **cc2WireframeModel** shape using the polarity reversal flag in this interface is the same as that of changing the polarity reversal flag that is intrinsic to the **cc2Wireframe** class. Updating one updates the other. Therefore, it is not possible to *double reverse* the polarity.

Decomposing a **cc2Wireframe** shape does not generally preserve the polarity reversal flag setting, unless the wireframe is decomposed as part of a shape tree model. Decomposing a **cc2WireframeModel** always preserves its shape model properties.

Directly decomposing a shape of type **ccGenPoly** or **cc2Wireframe** always results in a shape within which rounded corners are modeled by shapes of type **ccEllipseArc2**. The sign of the corner rounding is completely disregarded. However, if a shape of type **ccGenPolyModel** or **cc2WireframeModel** is directly decomposed, negatively rounded corners are converted into shapes of type **ccEllipseArc2Model** with an explicit weight of 0, thereby encoding the fact that vision tools should not model negatively rounded corners. Positively rounded corners are modeled by shapes of type **ccEllipseArc2**.

A more general rule is applied to shapes of type **ccGenPoly** or **cc2Wireframe** that are embedded within a shape tree: following decomposition of such a shape tree, negatively rounded corners for an embedded **ccGenPoly** or **cc2Wireframe** shape are modeled by a **ccEllipseArc2Model** with a weight of 0, provided that it or one of its ancestors has been assigned explicit model properties. Otherwise, all rounded corners are simply modeled as shapes of type **ccEllipseArc2**.

As decomposition is often employed for clipping and mapping, the same general rule is also applied for these operations. Note that a side effect of this behavior is that decomposed, clipped, or mapped shapes may contain contour tree shapes that have explicit, defaulted shape model properties, even though the original generalized polygon shapes, from which the contour trees were decomposed, did not.

As most vision tools interpret shapes only after assigning explicit shape model properties, this difference in decomposition behavior is only a consideration when you want to directly examine or use a decomposed shape. Vision tools, which may decompose shapes internally, will interpret negative corner rounding correctly, even when no explicit model properties have been assigned, unless otherwise noted.

Effect of Shape Manipulation on Model Properties

Operations that result in new shapes for which effective shape model properties are unchanged from the original shape *preserve* shape model properties. That is, the contours defined by the new shape have the same effective weight, polarity, magnitude, and ignore polarity flag values as in the original shape.

Having the same effective polarity means that the positive and negative sides of contours defined by the shape are unchanged relative to the spatial relationships between the contours themselves. For example, if manipulating a region shape that is positive on the inside produces a new region shape that is also positive on the inside, then polarity has been preserved.

In general, most operations on shapes preserve shape model properties. You can avoid violations of this general rule by explicitly specifying shape model properties for any shape prior to manipulation. For shape trees, explicit model properties need be specified only for the root to ensure shape model preservation.

Note: CVL vision tools, as a rule, internally assign explicit shape model properties before any shape manipulation, and therefore always interpret the intended polarities of shapes correctly. You do not need to assign model properties explicitly in your code before using shapes with vision tools.

Exceptions to these general rules of shape model property preservation for various types of shape manipulation are noted in the next sections.

See the *Shapes* chapter of the *CVL User's Guide* for more information on shape manipulation functions.

Decomposing

Decomposing a shape using **decompose()** always preserves shape model properties when explicit shape model properties have been assigned. If shape model properties have not been explicitly assigned, **decompose()** will preserve shape model properties for all but **cc2Wireframe** shapes.

Although the polarities of individual boundaries remain unchanged after decomposition, shape decomposition does not conserve topological properties (for example, the hole relationships of a region tree). What is *inside* and what is *outside* of a shape may be completely different following decomposition. Therefore, the polarity of the inside of a shape may be different from the polarity of the inside of its decomposition. For example, setting the inside polarity of an ellipse annulus to positive and then decomposing it will produce a general shape tree with two concentric contour trees, the outer contour tree having positive inside polarity and the inner contour having negative inside polarity. The decomposed shape will have inconsistent inside polarities although the polarities of the contours themselves will be equivalent to the polarity of the annulus.

Clipping

Clipping a shape using **clip()** always preserves shape model properties when shape model properties have been explicitly assigned. When shape model properties have not been assigned, **clip()** will preserve shape model properties for all shapes except **cc2Wireframe**s, for which any polarity reversals are stripped away when the clipping region actually intercepts the shape.

As clipping region shapes may produce non-region shapes (for example, open contours), the polarity of the inside of a shape may no longer have the same meaning as it did prior to clipping.

Reversal

Reversing a shape using **reverse()** always preserves weights, magnitudes, and ignore polarity flags. However, the effective polarities of the contours in the resulting shape will always be reversed from those of the original shape. If this effective polarity reversal is not desired, you can toggle the polarity reversal flag for the shape after calling **reverse()**. This two-step process of first calling **reverse()** and then toggling the polarity reversal flag has the effect of preserving shape model properties.

Mapping

Mapping a shape using the **map()** functions provided by any of the terminal shape classes generally does not preserve shape model properties. This operation removes any shape model properties associated with the shape, resulting in a shape that is no longer a shape model object. The exception to this rule is pure wireframes, or **cc2Wireframe**s that are not **ccShapeModel**s (wireframes for which only the polarity reversal property is specified). Mapping pure wireframes with **map()** does not remove polarity information.

Mapping using **mapShape()** generally preserves shape model properties. In fact, if model properties have been explicitly specified for a shape, **mapShape()** always preserves the properties defined within that shape. Again, the exception is pure wireframes. Mapping pure wireframes with **mapshape()** will remove polarity information.

If shape model properties have not been explicitly specified for a shape, shape model properties may not be preserved when mapping non-region-tree shapes by a transform having a rotation matrix with a negative determinant. In this case, shape polarities toggle for shapes that undergo a handedness flip as a result of the transformation. Polarity changes can be somewhat unpredictable when mapping any non-tree-model shapes (including region trees) having children or boundaries or both, with explicit model properties.

Flattening

Flattening shape trees using **flatten()** always preserves shape model properties when shape model properties have been explicitly specified for the root node. Otherwise, shape model properties are, in general, not preserved.

Connecting

Connecting a general shape tree using **connect()** always preserves shape model properties when shape model properties have been explicitly specified for the root node. Otherwise, shape model properties are, in general, not preserved.

Flipping a Region Tree

Flipping a region tree with **flip()** has the effect of reversing the polarity of all contours within the region tree while preserving the other properties (weight, magnitude, and ignore polarity flag). Any explicitly set polarity reversal flags remain unchanged.

Note: The inside polarity of a flipped region tree remains unchanged because of the compound effect of flipping and the effective polarity reversal. If the effective polarity reversal is not desired, you can toggle the polarity reversal flag for the shape.

Using Shape Models

CVL provides type definitions for all shape models. For example, **ccRectModel** is a type definition for the **ccShapeModelTemplate<>** class when instantiated with a **ccRect** argument as follows:

```
typedef ccShapeModelTemplate<ccRect> ccRectModel;
```

The following are the available shape model type definitions in CVL:

• **ccGeneralShapeTreeModel**	• **ccAffineRectangleModel**
• **ccContourTreeModel**	• **ccGenPolyModel**
• **ccRegionTreeModel**	• **cc2WireframeModel**
• **cc2PointModel**	• **ccBezierCurveModel**
• **ccFLineModel**	• **ccDeBoorSplineModel**
• **ccLineModel**	• **ccInterpSplineModel**
• **ccLineSegModel**	• **ccHermiteSplineModel**
• **ccRectModel**	• **ccGenRectModel**
• **ccCircleModel**	• **ccGenAnnulusModel**
• **ccAnnulusModel**	• **ccPolylineModel**
• **ccEllipse2Model**	• **ccEllipseAnnulusSectionModel**
• **ccEllipseAnnulusModel**	• **ccEllipseArc2Model**

You can use these shape models and shape tree models, rather than wireframe shapes, for PatMax shape training.

Model Maker

This section provides information about using the Model Maker feature included in CVL.

Note: This section provides an overview of the capabilities and features of Model Maker. For detailed information on using Model Maker, see the header files *ch_cvl\extract.h*, *ch_cvl\polarize.h*, and *ch_cvl\snap.h*.

Model Maker Overview

The CVL Model Maker is a collection of three tools that let you extract, manipulate, and refine feature shapes from images. You can use these extracted shapes to train a PatMax pattern. Model Maker comprises three related tools:

- The Contour Extraction tool extracts image features and constructs contours that represent the image features.
- The Shape Polarization tool takes one or more contours and sets their polarities based on a supplied image. The tool can polarize both extracted contours and contours that you construct synthetically
- The Shape Snapping tool takes a reference shape and adjusts it to fit the features in a supplied image.

Both the Contour Extraction tool and the Shape Snapping tool can work with grey-scale input images or with featurelet chains that you have extracted from other images.

Extracting Feature Shapes

Shape extraction is a mechanism that automatically creates CVL shapes based on the contours of an image.

Shape Extraction and Coordinate Spaces

All shape operations take place in the client coordinate system of the input image. When you specify dimensional tolerance and threshold values, such as the minimum or maximum chain length, the values are interpreted in client coordinates, not image coordinates.

How Shape Extraction Works

Shape extraction begins by finding the edges in an image. Edges whose pixel difference is within the contrast threshold and whose end points are within the connection tolerance are joined together into chains. In general you will not need to change the connection tolerance unless your image has artifacts that cause gaps in edge chains that should otherwise be joined together. Connection tolerance is always expressed in client coordinate units.

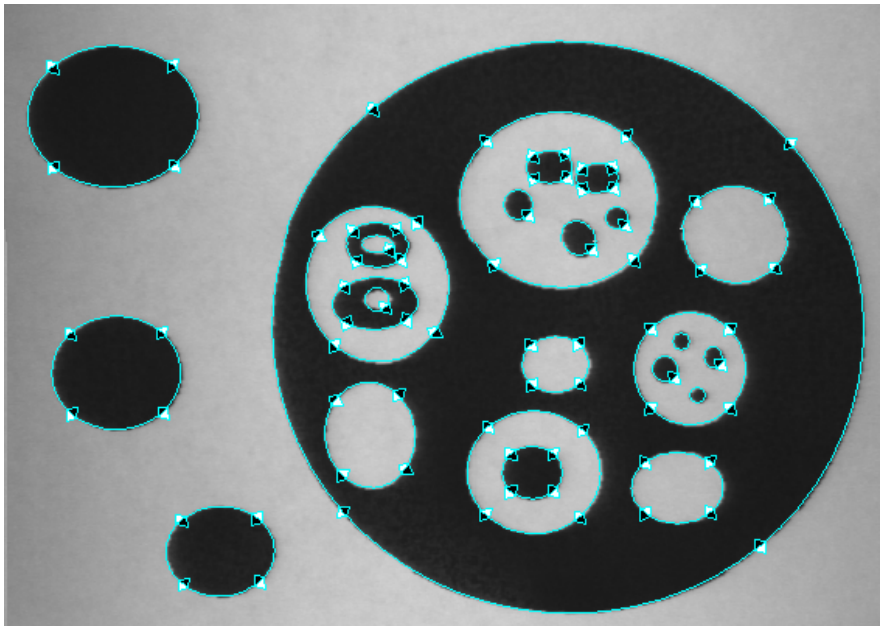
If perimeter limits are enabled, chains whose total length is less than the minimum or greater than the maximum perimeter are discarded. Perimeters are expressed in client coordinate units.

Once the shape extractor finds all the edge chains in the image, it uses a shape vocabulary of primitive shapes to convert that edge chain information into CVL shapes. The shape vocabulary does not specify the kinds of shapes that the shape extractor extracts. Instead, the vocabulary specifies the CVL shapes into which the extractor converts the edge chains that it finds. The shape vocabulary must contain at least one of the following:

- Line segment
- Circle
- Ellipse
- Circular Arc
- Elliptical Arc

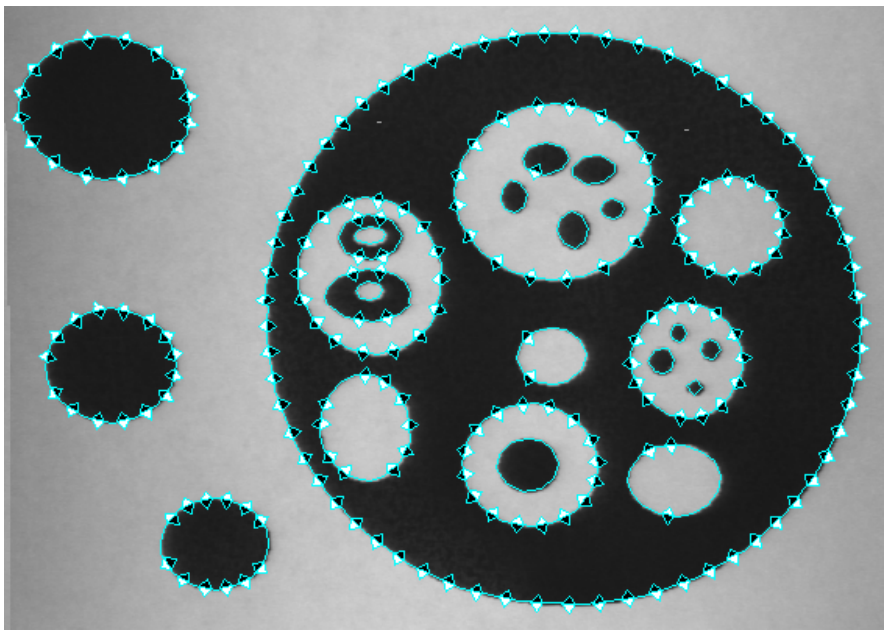
When possible, the shape extractor converts the edge chains into one of the primitive vocabulary shapes. Otherwise, if the vocabulary contains arcs or line segments, the chains are converted into contours that contain arcs and line segments.

For example, if the vocabulary consists only of circles and ellipses, and you try to extract shapes from the following image, the shape extractor converts the edge chains into circles and ellipses.



Extracted shapes

If you were to use the same image and change the vocabulary so that it included only line segments, the shape extractor would find the same shapes, but it would use contours composed of short line segments to trace the edges of the circles and ellipses.



Extracted shapes

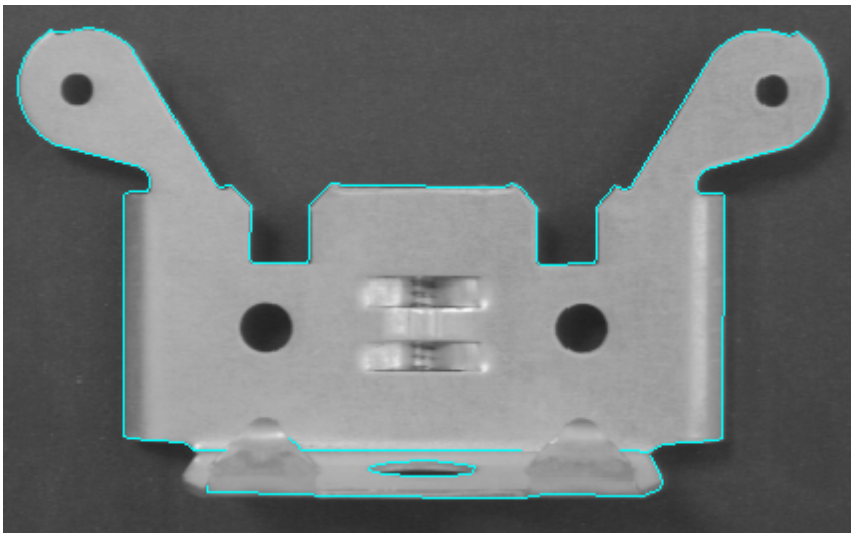
Note: The diamond graphics are not part of the extracted shapes; they indicate the relative number of shapes extracted.

Controlling Shape Extraction

The shape extractor uses an approximation tolerance to control how it converts the edge chains into CVL shapes. The approximation tolerance is the upper limit on the deviation between the edge chains and the extracted shape. Smaller

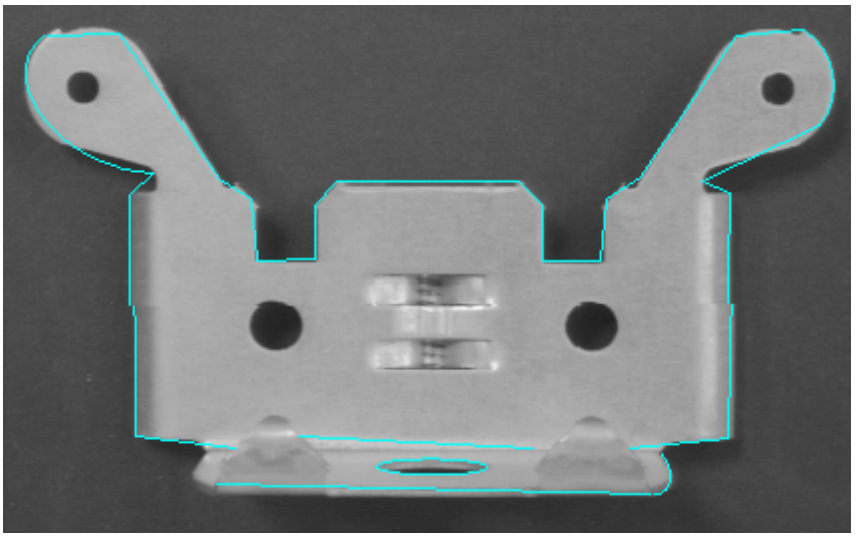
values result in shapes that follow the edges more closely, but higher values result in simpler shapes. Approximation tolerance is always expressed in client coordinate units.

For example, the figure below shows the result of a shape extraction with the approximation tolerance set to 0.7.



Extraction with small approximation tolerance

The figure below shows the result of shape extraction on the same image but with the approximation tolerance set to 6.0.



Extraction with large approximation tolerance

Constraining Shape Extraction

To make shape extraction more manageable, you can impose certain limits on the kinds of shapes that the shape extractor returns:

- Limit extraction by the minimum and maximum perimeter length
- Limit extraction to open or closed shapes or either
- Limit extraction by shape polarity (dark on light or light on dark) or ignore polarity
- Limit the number of shapes to extract

If you limit extraction to open shapes (or if you do not care whether the extracted shapes are open or closed) the vocabulary must include at least one open shape primitive: Line Segment, Circular Arc, Elliptical Arc. Closed shapes can be constructed out of open shape primitives.

The shape extractor ranks the shapes it finds using one of three criteria. By default the extractor ranks shapes by perimeter. You can also specify that it rank the extracted shapes by contrast or by proximity to a shape that you specify. The rank criterion determines the order in which the extracted shapes are returned. If you limit the number of shapes to return, shapes beyond the number you specified are discarded after they have been ranked.

Shape Polarization

Unlike shape extraction, shape polarization takes a CVL shape that you supply, along with an image and the pose at which align the shape to the image, and sets the polarity of the shape based on the data in the image.

How Shape Polarization Works

The Shape Polarization tool works by measuring the image polarity along a series of probes which are normal to the shape being polarized. You can specify the frequency of the probes (how closely spaced they are along the contour), the length of the probes (how far from the contour being polarized to search for an edge), and the contrast threshold (how strong an edge is required to polarize the shape).

Determining Effective Polarity

All CVL shapes have an implicit polarity (positive, negative, or indeterminate) prior to being polarized. The effective polarity of a shape after polarization is determined by both the initial polarity and the polarity detected in the image.

If both polarities are indeterminate, the resulting polarity is also indeterminate. If one polarity is indeterminate, then the resulting polarity is set to the determinate polarity (positive or negative). If both polarities are set and they are the same, the result is the same as the initial polarities. If the polarities are both set but do not match, then you can control how the final polarity is set (to the initial shape polarity, to the image polarity, or to indeterminate polarity).

Refining Shapes from Image Features (Shape Snapping)

Shape snapping is similar to shape extraction in that feature information is extracted from a supplied input image (or featurelet set). Unlike shape extraction, shape snapping uses the extracted feature information to refine the geometry of one or more shapes that you supply. Typically, you use shape snapping to refine the geometry of synthetically generated shapes based on image data.

As with shape polarization, shape snapping requires that you supply an input image and the pose at which to align the shape to the image; the Shape Snapping tool does not attempt to align the shape with the image.

How the Shape Snapping Tool Works

You operate the Shape Snapping tool by supplying the input shape to snap and specifying the type of the output shape. The output shape types supported by the Shape Snapping tool are listed below:

- Line and line segment
- Circle and circular arc
- Ellipse and elliptical arc
- Rectangle
- Polyline

You can either specify the type of output shape explicitly, or you can direct the tool to automatically determine the shape type.

Fitting Tool

This chapter describes the Fitting tool, a tool that fits sets of points to other sets of points, sets of lines, single lines, circles, and ellipses.

This chapter contains the following major sections:

Some Useful Definitions on page 42 lists definitions you will find useful while reading this chapter.

Fitting Tool Overview on page 42 describes the purpose of the Fitting tool.

How The Fitting Tool Works on page 42 provides a description of the point fitting procedure.

Using The Fitting Tool on page 46 describes Fitting tool classes and global functions you would use in an application.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

degree of freedom: Transformation that can be characterized by a single numeric value

least-squares method: A fitting technique that considers the best fit to be the one that produces the lowest total value for the squares of the distances between the features

minimum error method: A fitting technique that considers the best fit to be the one that produces the smallest maximum distance between any one point and the feature being fitted

transformation: Mathematical representation of the equations that describe the conversion of points from one coordinate system to another coordinate system

Fitting Tool Overview

The Fitting tool performs two basic functions:

- Determines the line, circle, or ellipse that best fits a set of points that you specify
- Determines the transformation that best fits one set of points to another set of points, or a set of points to a set of lines

You use the Fitting tool to simplify the task of developing information about the location and orientation of objects based on individual point locations that you determine using other vision tools.

How The Fitting Tool Works

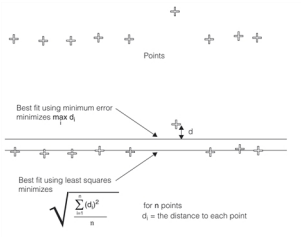
The Fitting tool is a collection of CVL routines for fitting points to geometric shapes, and for determining transforms between given point sets or a point set and given lines. The following sections describe how the Fitting tool works in each of these applications.

Fitting a Line

The Fitting tool lets you fit a line to a set of points using either of the following two methods:

- Minimizing the sum of the squares of the distances between all the points and the fit line
- Minimizing the maximum distance between any one point and the fit line

The figure below shows a set of points and the lines that offer the best fit to the points using both methods.



Fitting a line to points using least squares and minimum error methods

Threshold Error

When you fit a line to a set of points with the Fitting tool, you can specify an error threshold value whose meaning depends on the fit method being used. The tool will only indicate that it has fit a line (*found* = true) if the error calculated by the Fitting tool is less than the threshold you specify. When the error threshold is exceeded you can still obtain the fitted line from the result if you wish.

When you are using the least squares method, you specify a maximum least squares value as the threshold. With the minimum error method, you specify a maximum point-to-line distance.

Discarding Outlying Points

You can specify that a number of the points you supply to the Fitting tool be discarded. The figure below shows one point discarded by the Fitting tool when you supply 9 points and specify that the tool discard one of them. To determine which point to discard, the Fitting tool fits all possible subsets and keeps the one that produced the best score.

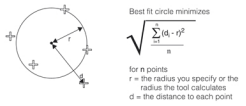
Discarding an outlying point

The tool always discards the number of points you specify.

Fitting a Circle

The Fitting tool lets you fit a circle to a set of points using the least squares method. You have the option of specifying the radius of the circle or having the Fitting tool calculate the radius. You also can specify a threshold value that the calculated fit error cannot exceed and a number of outlying points to be ignored.

To fit a circle, the Fitting tool minimizes the sum of the squares of the distances between the points and the circle, as shown in the figure below.



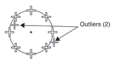
Computing the best fit circle

Threshold Error

When you fit a circle to a set of points with the Fitting tool, you can specify an error threshold value. The tool will only indicate that it has fit a circle (*found* = true) if the error calculated by the Fitting tool is less than the threshold you specify. When the error threshold is exceeded you can still obtain the fitted circle from the result if you wish.

Discarding Outlying Points

You can specify the number of points in the point set that the Fitting tool discards. The figure below shows the two points discarded by the Fitting tool when you supply 10 points and specify that the tool discard two points To determine which points to discard, the Fitting tool fits all possible subsets and keeps the one that produced the best score.



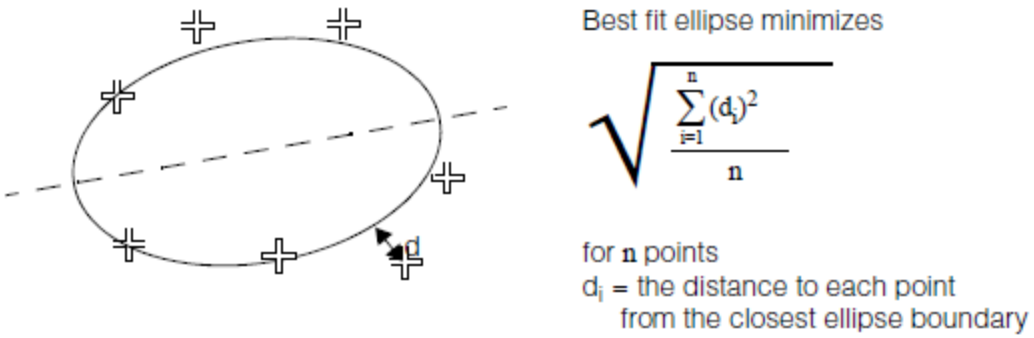
Discarding outlying points

The tool always discards the number of points you specify.

Fitting an Ellipse

The Fitting tool lets you fit an ellipse to a set of points using the least squares method. You have the option of specifying the ellipse orientation as the angle of either ellipse axis. If you do not specify an axis angle, the Fitting tool locates the ellipse automatically. You also can specify a threshold value that the calculated fit error cannot exceed and a number of outlying points to be ignored.

To fit an ellipse, the Fitting tool minimizes the sum of the squares of the distances between the points and the ellipse boundary, as shown in the figure below.



Computing the best fit ellipse

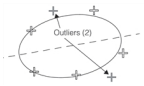
Threshold Error

When you fit an ellipse to a set of points with the Fitting tool, you can specify an error threshold value. The tool will only indicate that it has fit an ellipse (*found* = true) if the error calculated by the Fitting tool is less than the threshold you specify. When the error threshold is exceeded you can still obtain the fitted ellipse from the result if you wish.

When you fit an ellipse to a set of points with the Fitting tool, you can specify an error threshold value. The tool will only indicate that it has fit the ellipse if the minimum sum of squares value shown in the previous section is less than the threshold you specify.

Discarding Outlying Points

You can specify the number of points in the point set that the Fitting tool discards. The figure below shows the two points discarded by the Fitting tool when you supply 7 points and specify that the tool discard two points To determine which points to discard, the Fitting tool fits all possible subsets and keeps the one that produced the best score.



Discarding outlying points

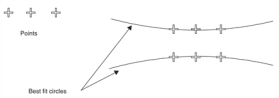
The tool always discards the number of points you specify.

Singular and Unstable Solutions

To enable the Fitting tool to fit a line to a set of points, you must define at least two distinct points; to enable the Fitting tool to fit a circle to a set of points, you must define at least three distinct points.

Even if you specify the minimum number of points, it is possible for the Fitting tool to be unable to fit a line or circle to a set of points. The most common cause for such a *singular solution* is that the points are very close to each other.

Another consequence of poorly located points can be an *unstable solution*. An unstable solution is one in which two or more extremely different solutions offer nearly identical values for the sum of the squares of the distances. The figure below shows a set of three points for which the solution is unstable. Even though the minimum number of points is specified, there are two extremely different circles that fit the points equally well.



Unstable solution (fitting a circle to three points)

Fitting Points to Points

The Fitting tool can determine the 3, 4, 5, or 6-degree-of-freedom transformation that best describes the relationship between two point sets. You order the points in set 1 and set 2 such that they correspond in numerical order. Point 1 in set 1 corresponds to point 1 in set 2, point 2 to point 2, and so on.

The Fitting tool considers the *best fit transformation* to be the transformation that, when applied to the first set of points produces a set of points which minimizes the sum of the squares of the distances between each transformed point and the corresponding point in the second set of points.

The figure below shows how the Fitting tool computes the best fit transformation for two sets of four points. After applying the best fit transformation to the first set of points (shown in black), the sum of the squares of the distances between each of the points in the first set and the corresponding point in the second set (shown in grey) is minimized.

Best fit transformation for two sets of points

When the Fitting tool calculates the transformation that best describes the relationship between the two point sets, it does so based on the number of degrees of freedom that you specify.

Fitting Points to Lines

The Fitting tool can determine the 3, 4, 5, or 6-degree-of-freedom transformation that best describes the relationship between a set of points and a set of lines. You order the point set and the line set such that they correspond in numerical order. Point 1 corresponds to line 1, point 2 to line 2, and so on.

The Fitting tool considers the *best fit transformation* to be the transformation that, when applied to the set of points produces a set of points which minimizes the sum of the squares of the distances between each transformed point and the corresponding line.

The figure below shows how the Fitting tool computes the best fit transformation for a set of four points and a set of four lines. After applying the best fit transformation to the set of points (shown in black), the sum of the squares of the distances between the each of the points in the first set and the corresponding line (shown in grey) is minimized.

Best fit transformation relating a point set to a line set

When the Fitting tool calculates the transformation, it does so based on the number of degrees of freedom that you specify.

Specifying Multiple Points per Line

In addition to specifying a separate line for each point, as shown in *Best fit transformation relating a point set to a line set* on page 45, you can also specify multiple points for a single line. When you specify multiple points per line, the Fitting tool computes the best fit transformation by considering each distinct point-to-line correspondence that you specify.

The Fitting tool provides functions that let you fit four groups of points to a rectangle. You specify the rectangle and four sets of points, each set corresponding to a side of the rectangle. The Fitting tool returns the transformation that produces the best fit considering all the point-line correspondences. See the figure below.

Best fit transformation relating a point set to a rectangle line set

In this example, the two horizontal lines in the rectangle each have three corresponding points and the two vertical lines in the rectangle each have two corresponding points. Each point group in the diagram is identified by a shaded background box.

You can specify both single point-line pairs *and* correspondences between a set of points and a single line at the same time. For example, you could specify the correspondences shown in *Best fit transformation relating a point set to a line set* on page 45and the correspondences in the figure above in a single tool instance, and then run the tool on all the correspondences. The Fitting tool returns the transformation that produces the best fit considering all the correspondences.

Using The Fitting Tool

The Fitting tool is a collection of CVL routines for fitting points to geometric shapes, and for determining transforms between given point sets or a point set and given lines. Each application of the tool has its own API. The following sections describe the API for each application.

Line Fitter

The following classes and global function comprise the line fitter API for the Fitter tool:

Class/Function	Description
cfLineFit()	The global function you call to fit a set of points to a line. **ccLineFitParams** and **ccLineFitResults** objects are passed as parameters. The input points are passed in as a *vector<cc2Vect>*. The found line is returned in **ccLineFitResults** as a line of type **ccFLine**.
ccLineFitDefs	Contains the **fit_mode** enum that defines the tool fit mode.
ccLineFitParams	Contains the run time parameters.
ccLineFitResults	When the tool runs it fills in this object with the results.

Circle Fitter

The following classes and global function comprise the circle fitter API for the Fitter tool:

Class/Function	Description
cfCircleFit()	The global function you call to fit a set of points to a circle. **ccCircleFitParams** and **ccCircleFitResults** objects are passed as parameters. The input points are passed in as a *vector<cc2Vect>*. The found circle is returned in **ccCircleFitResults** as a circle of type **ccCircle**.
ccCircleFitDefs	Contains the **fit_mode** enum that defines the tool fit mode.
ccCircleFitParams	Contains the run time parameters.
ccCircleFitResults	When the tool runs it fills in this object with the results.

Ellipse Fitter

The following classes and global function comprise the ellipse fitter API for the Fitter tool:

Class/Function	Description
cfEllipseFit()	The global function you call to fit a set of points to a ellipse. **ccEllipseFitParams** and **ccEllipseFitResults** objects are passed as parameters. The input points are passed in as a *vector<cc2Vect>*. The found ellipse is returned in **ccEllipseFitResults** as a ellipse of type **ccEllipse**.
ccEllipseFitDefs	Contains the **fit_mode** enum that defines the tool fit mode.
ccEllipseFitParams	Contains the run time parameters.
ccEllipseFitResults	When the tool runs it fills in this object with the results.

Point to Point Fitter

The point to point fitter is encapsulated in the class **ccLSPointToPointFitter**. When you use the Fitter tool this way you provide two sets of points, and the tool computes a transform you can then use to transform points from one space to the other. You can provide the point sets when you construct the tool, or you can default construct the tool and specify the point sets later. You call **ccLSPointToPointFitter::fit()** to run the tool and the function returns the computed transform. You can specify new point sets and reuse the tool object to compute other transforms.

The following are some of the more important member functions:

Member function	Description
fit()	Run the tool using the currently defined point sets.
reset()	Reset the tool to its default constructed state.
update()	Specify two new point sets.

Point to Line Fitter

The point to line fitter is encapsulated in the class **ccLSPointToLineFitter**. When you use the Fitter tool this way you provide a set of points and a corresponding set of lines. When you run the tool it computes a transform between the point space and the line space. Lines can be specified as individual **ccFLine** objects or as a **ccRect** rectangle which defines two horizontal lines and two vertical lines.

You can define the points and lines when you construct the tool, or you can default construct the tool and specify them later. You call **ccLSPointToLineFitter::fit()** to run the tool and the function returns the computed transform. You can specify new point sets and reuse the tool object to compute other transforms.

The following are some of the more important member functions:

Member function	Description
fit()	Run the tool using the currently defined points and lines.
reset()	Reset the tool to its default constructed state.
update()	Specify new points and lines.
fitRect()	Run the tool using the points and the rectangle passed in as parameters.

Classifier Tool

This chapter describes the Classifier tool, a tool that scores and classifies input data values.

This section and *Some Useful Definitions on page 48* give an overview of the chapter and define some terms you will encounter as you read.

Classifier Tool Overview on page 48 provides an overview the Classifier tool.

How The Classifier Tool Works on page 51 provides a description of how the Classifier tool works.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

feature vector: An n-dimensional vector describing a single collection of input values.

scoring function: A function that maps an input feature value to a normalized score.

rule: A list of scoring functions, one function for each dimension of the feature vector being scored.

rule table: A list of rules, one rule for each category into which feature vectors are being classified.

Classifier Tool Overview

The Classifier tool scores and classifies features or objects based to the values of measurements of the features or objects. The Classifier tool is based on the following objects:

- Feature vectors, which contain one or more measurements of a feature or object
- Scoring functions, which map a single feature measurement into a normalized score
- Rules, which compute an overall score for a feature vector using a list of scoring functions
- Rule tables, which consists of one or more rules

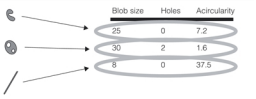
Each of these objects is described in this section.

Feature Vectors

A feature vector is an n-dimensional vector or point. A feature vector represents a collection of measurements or other data items about a single object. For example, if you are using the Classifier tool to classify features detected by the Blob tool, you might define the following 3-dimensional feature vector:

1. Blob size
2. Number of holes inside blob
3. Blob acircularity

The figure below shows three features detected by the blob tool and the feature vectors that describe them.



Feature vectors

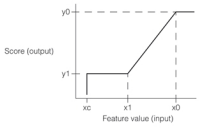
Each feature vector contains the same three items of information about each feature.

Note: For more information on the Blob tool, see the chapter *Blob on page 214*.

Scoring Function

A scoring function is a simple function that maps an input value to an output score. You define a scoring function for each dimension or element of the feature vector you will be classifying.

You define a scoring function by specifying low and high input and output values; the Classifier tool creates a scoring function based on the values you supply. The figure below shows a scoring function.



Scoring function

You define the scoring function by defining values for *xc*, *x1*, *x0*, *y1*, and *y0*. The Classifier tool uses the scoring function you define to map feature values to scores as follows, given that *x0* is greater than *x1* and *x1* is greater than *xc*.

- Feature values above *x0* are mapped to a score of *y0*.
- Feature values below *xc* are mapped to a score of 0.
- Feature values between *xc* and *x1* are mapped to a score of *y1*.
- Feature values between *x1* and *x0* are mapped linearly to the range of scores between *y1* and *y0*.

While input values can have any value, output scores must be in the range from 0.0 through 1.0.

The Classifier tool lets you create several types of scoring functions. For more information on scoring functions, see the section *Scoring Function Types* on page 51.

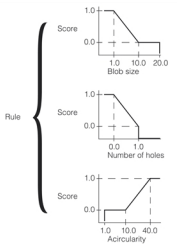
Rules

A rule is a set of scoring functions, one for each dimension of the feature vectors you are classifying. A rule computes an overall score for a single feature vector.

In the case of the example shown in *Feature vectors* on page 48, you would need to define a scoring function for each of the three dimensions of the feature vector to create a rule for classifying those features.

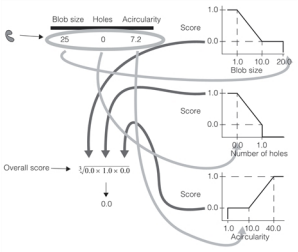
A rule computes an overall score for a feature vector by computing the geometric mean of the scores produced by each of the scoring functions that comprise the rule.

For example, if you knew that features with acircularity values above 10.0 were almost always scratches as long as they did not contain any holes and were not larger than about 10 pixels in size, you could define the set of scoring functions shown in the figure below to create a rule for scratches.



Rule for scratches

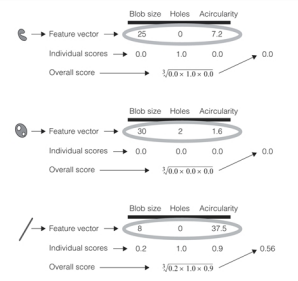
The rule shown in the figure above is intended to produce a high score for scratches and a low score for features that are not scratches. The figure below shows the scores produced by the rule for the first feature from *Feature vectors* on page 48.



Applying the rule to a single feature vector

For each dimension of the feature vector, the Classifier tool computes the score for that feature dimension. The overall feature score is the geometric mean of the individual dimension scores.

Applying the rule as shown in the figure above to all three of the features shown in *Feature vectors* on page 48 produces the overall scores shown in the figure below.



Applying the rule to each feature

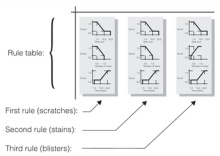
A shown in the figure above, the rule produces a score of 0.56 for the scratch and scores of 0.0 for the non-scratch features.

Rule Tables

For most applications of the Classifier tool, you define a rule for each category into which a feature vector can fall. The preceding section, *Rules* on page 49, showed how to create a single rule to describe the category of scratches. Using the same technique, you could define additional categories for the other types of features shown in *Feature vectors* on page 48.

For example, you might define rules for scratches, blisters, and stains. Each rule would use a different set of scoring methods to produce a high score for the type of feature.

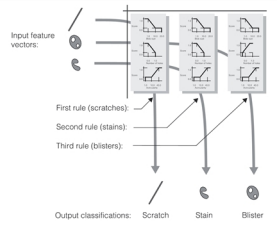
A rule table is a collection of individual rules, where each of the rules has the same number of dimensions as the feature vectors you will be classifying. The figure below shows a rule table.



Rules table

At run time, the Classifier tool classifies feature vectors using a rule table by applying each rule in turn to the supplied feature vector and returning the label of the first rule that produces a score above a threshold that you specify.

The figure below shows how a rule table is applied to classify the features shown in *Feature vectors* on page 48.



Rules table applied to input feature vectors

How The Classifier Tool Works

This section describes how the Classifier tool works.

Feature Vectors

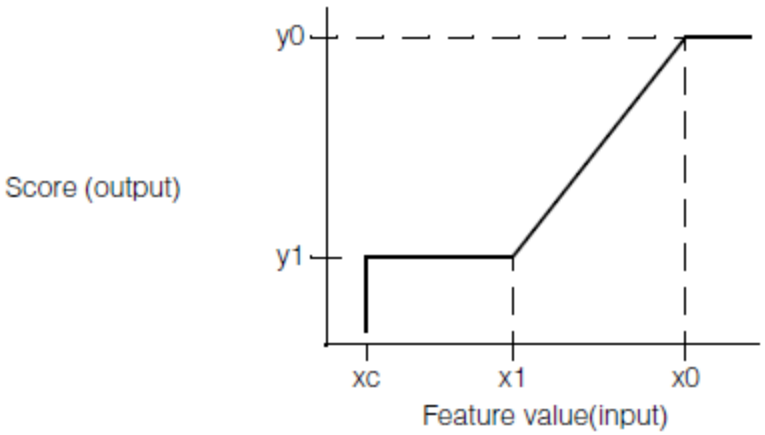
Feature vectors are implemented as Standard Template Library (STL) vectors of floating point values. You can only set the values of a feature vector when you construct the feature vector.

Scoring Function Types

Scoring functions can be one-sided or two-sided.

One-Sided Scoring Functions

One-sided scoring functions are defined by a low input value, high input value, cutoff input value and low and high output values, as shown in the figure below.



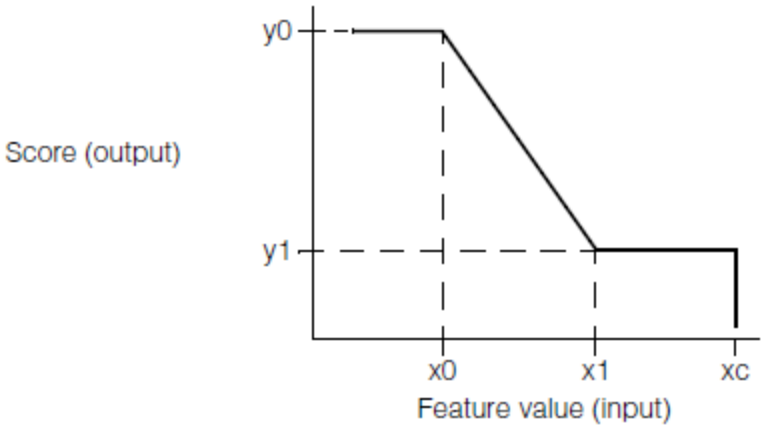
One-sided scoring function

You define the scoring function by defining values for *xc*, *x1*, *x0*, *y1*, and *y0*. The Classifier maps input values to score values as follows, given that *x0* is greater than *x1* and *x1* is greater than *xc*:

- Feature values above *x0* are mapped to a score of *y0*.
- Feature values below *xc* are mapped to a score of 0.
- Feature values between *xc* and *x1* are mapped to a score of *y1*.
- Feature values between *x1* and *x0* are mapped linearly to the range of scores between *y1* and *y0*.

The values you define for *y0* and *y1* must be between 0.0 and 1.0. You can specify any values for *xc*, *x1*, and *x0*, including negative values. You would specify negative values for one or more of the *xc*, *x1*, and *x0* points if you expected feature values less than zero.

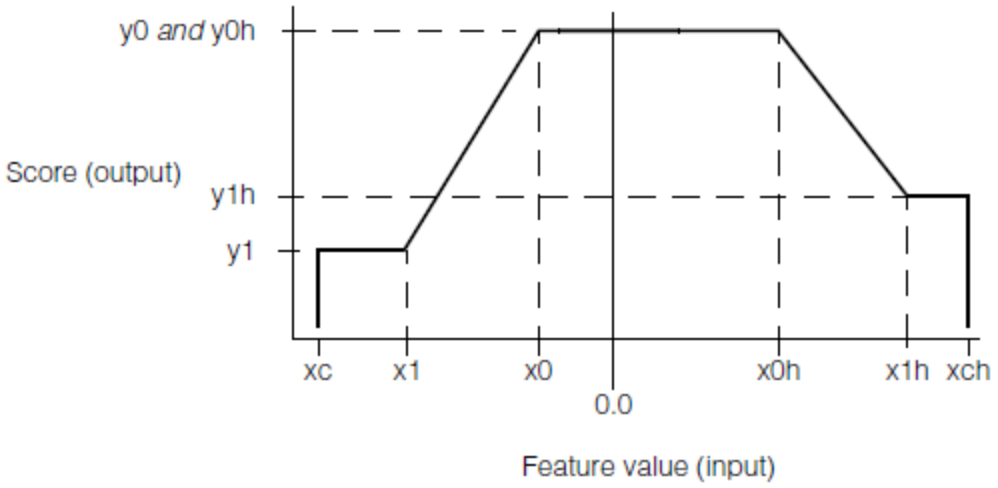
The figure below illustrates a one-sided scoring function which would be appropriate for cases where larger feature values should receive lower scores. You specify a function like this by supplying a value for *x1* that is greater than the value for *x0*.



Scoring function with *x1* greater than *x0*

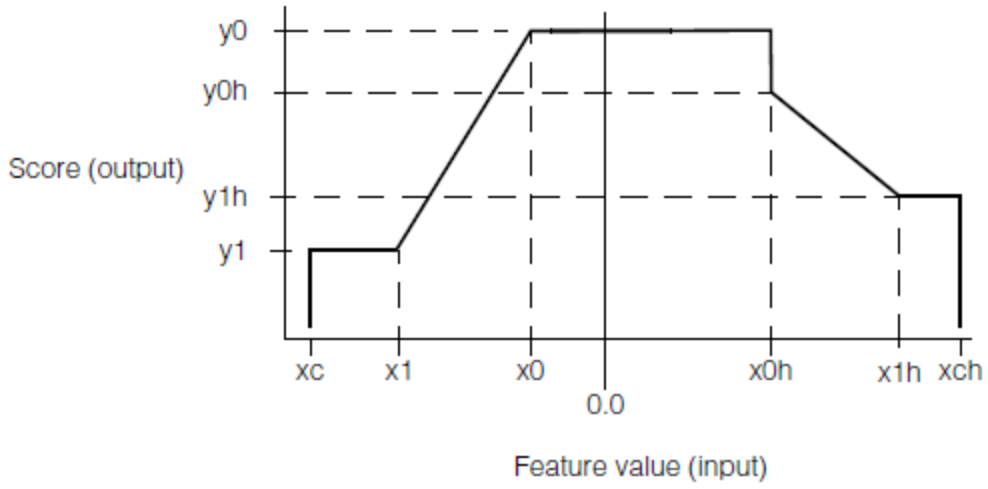
Defining a Two-Sided Scoring Function

A two-sided scoring function is composed of a pair of one-sided scoring functions. A two-sided scoring function lets you assign high scores to feature values within a particular range. The figure below shows a two-sided scoring function.



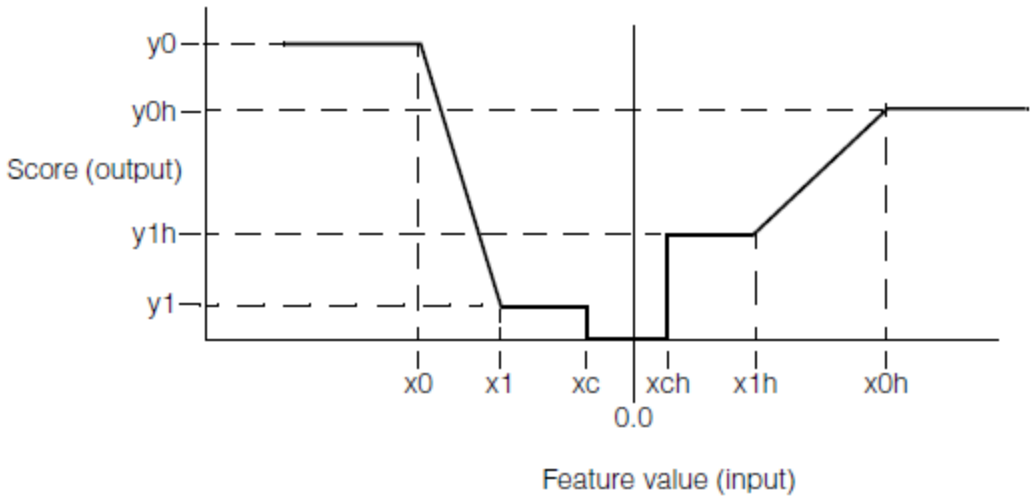
Two-sided scoring function

The figure above shows a two-sided scoring function with distinct values for *y1* and *y1h* but the same values for *y0* and *y0h*. While it is possible to specify different values for *y0* and *y0h*, the resulting scoring function contains a discontinuity at *x0b*. The figure below shows a scoring function where *y0h* is different than *y0*.



Two-sided scoring function

You can also specify a two sided scoring function for cases where feature values *outside* of a particular range receive higher scores. The figure below shows an example of this type of two-sided scoring function.



Two-sided scoring function with low output values in the center

Creating Rules

To create a rule, you simply supply an STL vector of scoring methods. You must supply a scoring method for each element of the feature vectors you wish to score, and you must supply the scoring methods in the same order as the elements of the feature vectors.

The rule object has a scoring function which computes the geometric mean of the scores received by each element of the feature vector.

In addition to specifying the vector of scoring methods, you must also supply a *label* for the rule. This label is returned by the Classifier tool when it classifies a feature vector.

Creating Rule Tables

When you create a rule table, you simply supply an STL vector of rules. You must supply a rule for each classification into which you want to categorize feature vectors. Each rule in the rule table must have the same number of scoring functions.

Classifying Feature Vectors

You classify a feature vector by supplying it to a rule table along with a threshold value. The Classifier tool will score the feature vector using each rule in the rule table, in the order in which the rules were supplied to the rule table. The Classifier tool returns the label of the first rule that produces a score value greater than the threshold value you supply.

Point Matcher Tool

This chapter describes the Point Matcher tool, a tool that matches sets of points.

This section and *Some Useful Definitions on page 55* give an overview of the chapter and define some terms you will encounter as you read.

Point Matcher Tool Overview on page 55 provides an overview of point matching.

How The Point Matcher Tool Works on page 55 provides a description of how the Point Matcher tool works.

Using The Point Matcher Tool on page 58 describes how to use the Point Matcher tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

RMS error: A measure of the variation between two sets of data. The RMS error is computed by taking the square root of the mean of the squares of the individual errors, where the individual error is the difference between two corresponding data items.

transformation: Mathematical representation of the equations that describe the conversion of points from one coordinate system to another coordinate system

Point Matcher Tool Overview

The Point Matcher tool matches a set of trained two-dimensional model points with a set of two-dimensional points derived from a run-time image. It returns two pieces of information:

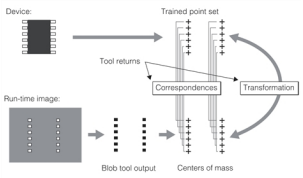
- The transformation that describes the difference between the trained points and the run-time points
- The correspondence between each trained point and a single run-time point (or an indication that no run-time point corresponded to the trained point)

The tool also returns information about how well the model points matched the run-time points.

Unlike most other alignment vision tools, the Point Matcher tool matches points rather than pixels or image features. Your application is responsible for extracting point information from input images, then passing this data to the Point Matcher tool.

A typical application for the Point Matcher tool would be to train a set of points corresponding to the tips of leads on an electronic device, then using the tool to determine the correspondence between the device and the locations of solder pads on a circuit board. (You might use the Blob tool to determine the locations of the pads on the board.)

The figure below shows how this application would work. Note that the tool returns both the transformation between the two point sets and the correspondence between each trained point and a run-time point.



Point Matcher tool application

How The Point Matcher Tool Works

This section describes how the Point Matcher tool works.

Training Model Points

You train the Point Matcher tool by supplying a vector of 2-dimensional points. The points are located in an implicit coordinate space (you do not need to specify any attributes of the coordinate space such as scale or handedness).

Matching Run-Time Points

At run time, you supply a vector of run-time points (which are located in the same implicit coordinate space as the trained points) along with a starting pose. The Point Matcher tool establishes a point-to-point correspondence between each point in the trained point set and a single point in the run-time point set, then computes the transformation between the two sets of points that minimizes the RMS error across all matched points.

The Point Matcher tool matches run-time points to trained points by following these steps:

- The tool maps the run-time points using the nominal or starting pose that you specify.
- The tool quantizes the locations of all trained points and all run-time points to a pair of grids (you specify the grid size), then computes the coarse transformation between the quantized point sets.
- If you specify a range of uncertainty for the scale, angle, or displacement of the run-time points from the nominal pose, the tool performs an exhaustive search across the entire search space that you specify to determine the coarse transformation between the trained points and the run-time points.
- Once the tool has computed the coarse transformation, the tool performs point-to-point correspondence matching using the unquantized point locations, then iteratively refines the coarse pose to minimize the RMS fit error between the two point sets.

Each of these steps is described in this section.

Mapping Points through the Nominal Pose

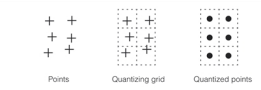
For most applications, you will have some idea of how the run-time points differ from the trained point set. If you have generated the run-time points by computing the centers of mass of features using the Blob tool, the point locations will be in the client coordinate system of the image you analyzed, while the trained points will be in whatever arbitrary coordinate system you defined for those points.

The Point Matcher tool lets you specify a nominal pose that describes this known difference between the trained points and the run-time points. The Point Matcher tool transforms each of the trained points using the nominal pose that you specify before it attempts to match the points to the run-time points.

Computing the Coarse Transformation

The Point Matcher tool computes the coarse transformation by quantizing the locations of all trained and run-time points to a pair of grids. Once the two sets of points have been quantized, the tool computes the grid-to-grid transformation between the trained and the run-time points.

The figure below shows how the tool uses a grid to quantize the point locations.



Quantizing points to a grid

Note: The Point Matcher tool uses a grid size that you specify (or you can configure the tool to automatically set the grid size to a reasonable value).

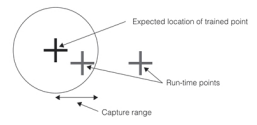
If the location, angle, or size of the run-time points can vary from point set to point set, you can specify one or more uncertainty ranges. The tool lets you specify the expected range of variation for x- and y-translation, angle, and scale. If

you specify an uncertainty range, the tool performs an exhaustive search across the entire search space that you specify to determine the coarse transformation.

Matching Points

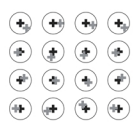
Once the Point Matcher tool has determined the coarse transformation between the trained points and the run-time points, it uses a separate procedure to refine that coarse pose. The tool applies the coarse transformation to the unquantized trained points. The resulting transformed points are the *expected point location*. The tool then selects the run-time point that corresponds to each expected point location and iteratively refines the coarse pose to reduce the RMS error across all matched trained points.

The tool considers each run-time point that lies within the *capture range* of the expected point location of a trained point to correspond to that point, as shown in the figure below.



Capture range

The process of matching points involves two basic steps: performing the point correspondence step for each trained point and computing the best fit for all points. The initial step is performed by matching each trained point to the run-time point that lies within the capture radius. The figure below shows the results of this iterative process for a set of 16 trained points and 16 run-time points (the trained points are shown in black while the run-time points are grey).



Minimize RMS error across all point correspondences

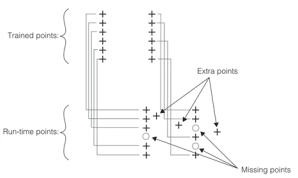
If more than one run-time point lies within the capture range for a single trained point, the Point Matcher tool computes the best fit across all trained points for both of the matching points. This generates two distinct results. The multiple results are ranked first by *coverage score* (the fraction of the trained points that had matching run-time points), and within results with matching coverage scores, by fit error (RMS error across all matched points).

Note: If there are many combinations of matching points, then the tool's operation can be slowed down considerably.

Missing and Extra Points

The Point Matcher tool is designed to handle cases where there are missing or extra points in the run-time point set. You can specify a minimum coverage threshold to the tool at run-time. If you specify a threshold, the tool will only return results where the ratio of trained points to matched points is greater than or equal to the threshold that you specify.

The figure below shows examples of missing and extraneous points. Note that the missing points are identified with a value of -1 in the correspondence list returned by the tool.



Missing and extra points

Point Matcher Tool Results

For each unique set of point correspondences, the Point Matcher tool returns the following information:

- A single transformation between the trained points and the run-time points
- The correspondence list (which trained point was matched to which run-time point)
- The coverage score (the ratio of the number of trained points to the number of matched trained points)
- The fit score (the RMS error across all trained points that were matched to run-time points)

Using The Point Matcher Tool

This section lists some general guidelines for using the Point Matcher tool.

Training Point Sets

In general, the Point Matcher tool is intended for use in matching point sets with between 10 and 150 points. The minimum number of trained points is 3, the maximum number is 10,000.

Run-Time Point Sets

For many applications, the run-time point set will contain more points than the trained point set. This is often caused by the presence of noise or extraneous or incidental features in the run-time image from which the run-time point set is derived. The Point Matcher tool is intended for use in matching run-time point sets with between 10 and 400 points. The minimum number of run-time points is 3, the maximum number is 20,000.

Run-Time Parameters

In addition to a run-time point set, you also supply the Point Matcher tool with the following run-time parameters:

- The nominal (starting) pose

  This pose should be your best estimate of the expected relationship between the trained point set and the run-time point set.

- One or more uncertainty ranges

  For each of the supported degrees of freedom (x-translation, y-translation, rotation, and scale), you should specify the expected range of variation, relative to the starting pose. The larger a search space you specify, the longer the tool will take to run.

- The grid size to use for point location quantization and coarse pose computation

  In general, you should specify a value of about 25% of the distance between the closest pair of points in the trained point set. (You can configure the tool to automatically compute this value for you.)

- The capture range to use for fine pose refinement

  In general, you should specify a value of about 25% of the distance between the closest pair of points in the trained point set. (You can configure the tool to automatically compute this value for you.)

- The maximum expected number of results

  For most applications, there is only one expected result. If, however, your application has more run-time points than trained points, you may need to obtain multiple results and examine the point correspondence lists to determine which result is correct.

Assigning Weights to Model Points

The Point Matcher tool supports the assignment of weights to specific model points using an overload of the **ccPointMatcher::train()** function. This overload has the following signature:

```
void train(const cmStd vector<cc2Vect>& modelPoints,
 const cmStd vector<c_Int32>& weights);
```

The first argument to this function, *modelPoints*, is the vector of model points to be trained. The second argument, *weights*, is a vector of weights to assign to each of the model points. The two vectors must have the same size, which must be at least three. The *weights* vector can contain 32-bit integer values that are either positive, negative, or zero, and it must contain at least one positive value.

Weights affect the calculation of the coverage score as follows:

- A match with a model point with a positive weight increases the coverage score.
- A match with a model point with a negative weight decreases the coverage score.
- A match with a model point with a weight of zero does not affect the coverage score.

If the Point Matcher tool is trained with weights, it computes the coverage score as the larger of zero or the ratio of the sum of the weights of matched model points to the sum of all positive weights. The minimum coverage value for the tool, in this case, must be less than or equal to the number of model points with positive weights to the total number of model points.

If the Point Matcher tool is trained without weights, the weight of each model point is one and the tool computes the coverage as the ratio of the number of matched model points to the total number of model points.

The **ccPointMatcher::weights()** getter retrieves the weights vector.

Image Stitching Tool

This chapter describes the Image Stitching tool, a tool that combines two or more source images into one composite result image. The tool is useful when a camera field of view is too small to capture the entire desired scene and multiple images are required. The tool can stitch together these images into a single image of the entire scene which can then be used by other vision tools.

This section and *Some Useful Definitions on page 60* provide an overview of the chapter and define some terms you will encounter as you read.

The chapter has the following major sections:

Image Stitching Tool Overview on page 61 introduces the Image Stitching tool and describes its purpose.

How the Image Stitching Tool Works on page 63 describes how the Image Stitching tool combines two or more source images into one composite image.

Using the Image Stitching Tool on page 64 describes how to use the Image Stitching tool in a C++ application.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

Blending mode: An operating mode of the Image Stitching tool where all source pixels are properly weighted to blend seamlessly into the result image. All valid pixels in all source images contribute to the result image. The order in which source images are stitched does not affect the result image.

bounding box: As used in this chapter, a bounding box is the smallest **ccPelRect** result image window that completely encloses all of the source image FOVs in client coordinate space.

client coordinate space: An internal Image Stitching tool working space that is common to all of the source images and to the result image. Each source image contains a unique calibration transform that maps its image into this common space. Conceptually, the Image Stitching tool builds the result in client coordinate space and the result image contains a transform that maps the client space result into the result image.

FOV: (Field Of View). The window (**ccPelRect**) into a source image (**ccPelBuffer**). The FOV window can specify any part of, or the entire image. The Image Stitching tool sees only the FOV window.

image stitching: The process of combining two or more images to make a single image. Source images typically overlap so that there are no blank areas in the result. Overlapping pixels are resolved in one of two ways. See *Blending* mode and *Overwrite* mode.

Overwrite mode: An operating mode of the Image Stitching tool where pixel values in the result image are directly replaced by corresponding pixel values in the current source image. Because current image pixels overwrite previous image pixels, the processing order of the source images will affect the result image. For regions in the common client space covered by more than one source image, the pixel values in the result image will be from the last source image stitched.

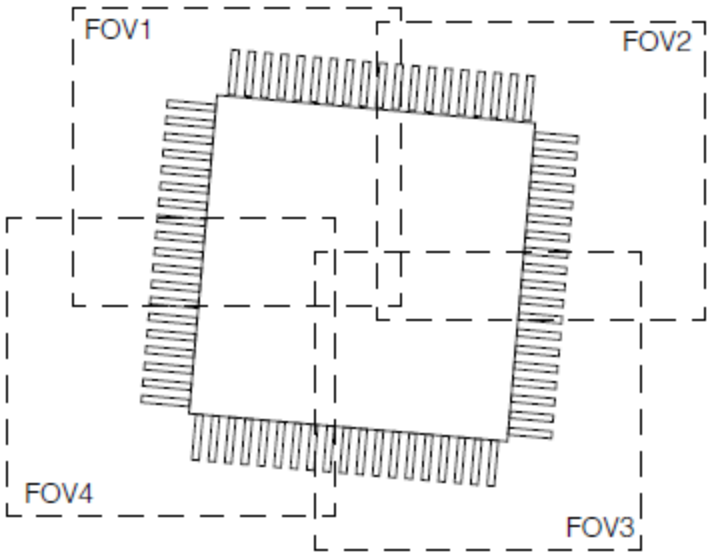
result image: As used in this chapter, this is the image produced by the Image Stitching tool.

scene: The entire view required by a vision application. As discussed in this chapter, a scene is too large to be captured by a single camera image, so the application is designed to capture the scene in multiple images which are then combined into one larger image using the Image Stitching tool.

source image: A single image used as an input to the Image Stitching tool. Some part of a scene.

Image Stitching Tool Overview

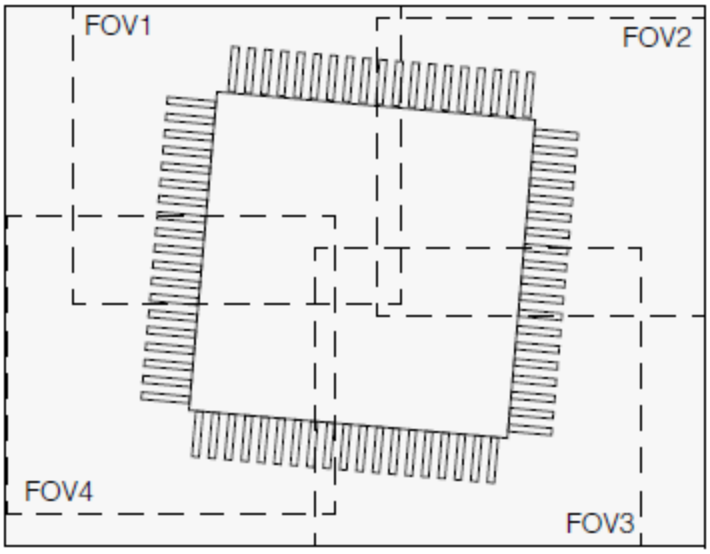
The Image Stitching tool combines two or more source images into one composite result image. The tool is useful when a camera field-of-view (FOV) is too small to capture an entire scene and multiple images are required. The tool can stitch together these images into a single image of the entire scene which can then be used by other vision tools. The figure below shows an example of a large-leaded SMD device captured in four images. Using the Image Stitching tool you can stitch these four images together to make a single result image of the entire device.



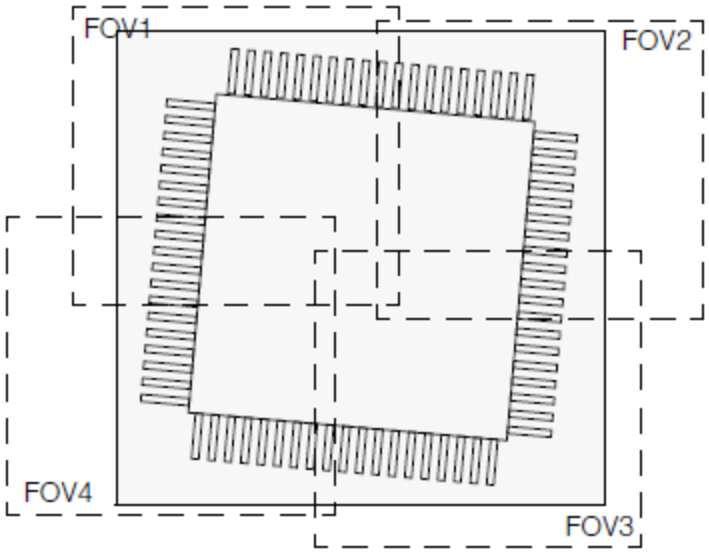
Example scene captured with four images

The source images can be provided by a single moving camera, by multiple fixed cameras, or by most any camera setup where each camera captures part of the same scene. Cameras need not be the same type or size but all cameras must be calibrated so that the client coordinate transform associated with each image targets a common client coordinate space.

The Image Stitching tool allows you to specify the result image window, or you can allow the tool to specify a default result window as the bounding box that encloses all of the source FOVs. *Result image window examples* on page 62 shows the default result image window (shaded area) for the example in the figure above, and also an example of a user-defined result window.



Default result image window



Example of a user-defined result image window

Result image window examples

Stitching Modes

The images to be stitched together should overlap one another so that all parts of the scene are covered by at least one FOV. In the overlap areas the Image Stitching tool must combine overlapped pixels to create the result image. For example, in *Result image window examples* on page 62 the same pixels near the center of the SMD device appear in three FOVs. Note that no pixels appear in all four FOVs since FOV2 and FOV4 do not overlap.

The Image Stitching tool combines these overlapping pixels according to the stitching mode you choose. Two modes are supported: *Overwrite* mode and *Blending* mode. In Blending mode, the overlapping pixels are blended together giving more weight to pixels that are closer to the center of their FOV. This produces a result image where the edges of the FOVs are less noticeable to the human eye; a smoother image.

The tool performs image stitching sequentially on the source images, in the order you specify. In Overwrite mode, overlapping pixels are not combined, but are overwritten as they are processed. For example, if a pixel in FOV1 is also

contained in FOV2, the FOV2 value will overwrite the FOV1 value. Thus, for all overlapping pixels, the result pixel value is from the last FOV processed.

Image Masks

Image masks can be used with both source images and result images.

Source Image Masks

When you pass a source image to the Image Stitching tool you have the option to specify an associated image mask. The image mask is aligned with the associated image in image space. There is a 1:1 correspondence between pixels in the image, and pixels in the mask. The mask pixel values are either 0 or 255. A 0 mask pixel value means the corresponding image pixel is ignored by the Image Stitching tool. A 255 mask value means the pixel is processed (stitched) by the tool. The mask allows you to be selective about which parts of a FOV are stitched into the result image.

Result Image Masks

The Image Stitching tool provides a result image and a corresponding result image mask. You can use the result image mask, or choose to ignore it. The mask is aligned with its associated result image in image space. There is a 1:1 correspondence between pixels in the result image, and pixels in the mask. The mask pixel values are either 0 or 255. A 255 mask value means the corresponding result image pixel was created (stitched) from the source images. A 0 mask pixel value means the corresponding result image pixel is empty. That is, there were no corresponding pixels from any source image that were stitched by the tool. The tool sets all empty pixels to a fixed value that you specify.

How the Image Stitching Tool Works

The Image Stitching tool accepts a series of input images, one at a time, and incrementally builds a result image from them. The tool works in a client coordinate space defined by the client coordinate transforms provided with each input image. There is no limit to the number of input images you can provide, however most applications use just a few, each image covering part of a scene that is too large for a single camera. The result image covers the entire scene and is typically a larger image made by stitching together all of the input images. The figure below shows an overview of the Image Stitching tool along with its inputs and outputs.

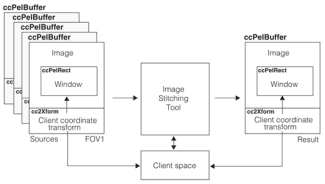


Image Stitching tool

Once initialized, the Image Stitching tool is programmed to process input images sent to it and build an output image from them, stitching the input images together into a composite output image.

Stitching

The Image Stitching tool builds the result image incrementally in client coordinate space while processing input images one at a time. The stitching process is performed pixel by pixel using source image pixels and corresponding client space pixels to create new pixels in client space. The following rules are used for processing (stitching) each pixel.

- The client space pixel location is found by using the source image pixel location transformed by the source image client coordinate transform.
- If a source image mask is provided with the source image, the mask pixel value must be 255 or the source pixel is skipped.

- If the client space pixel value is empty (not changed since initialization), the client space pixel value is set equal to the source image pixel value.
- If the client space pixel has been changed, the client space pixel value is updated according to the current Image Stitching tool mode, *Overwrite* mode or *Blending* mode.

Overwrite Mode: Pixel values in the result image are directly replaced by corresponding pixel values in the current source image. Because current image pixels overwrite previous image pixels, the processing order of the source images will affect the result image. For regions in the common client space covered by more than one source image, the pixel values in the result image will be from the last source image stitched.

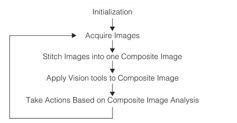
Blending Mode: All source images are properly weighted to blend seamlessly into the result image. All valid pixels in all source images contribute to the result image. The order in which source images are stitched does not affect the result image.

Using the Image Stitching Tool

The Image Stitching tool is a class of type **ccImageStitch** that you program to combine (stitch together) two or more source images into one composite result image. To use the tool you need to setup and calibrate your application, initialize the Image Stitching tool, run the tool on acquired images, and obtain the results. The following sections discuss each of these steps relative to using the Image Stitching tool in a C++ application.

Setup and Calibration

The Image Stitching tool is typically used in a vision application where multiple images are needed to capture a single scene that you wish to process with other vision tools. The following is a simplified diagram of such an application.



An important part of the application design is that all of the images you wish to combine using the Image Stitching tool must share a common client coordinate space. You will need to calibrate your cameras so that each image window maps into the proper client coordinate space area. An example of these mapped areas are the FOVs shown in *Result image window examples* on page 62. The calibration transforms that accomplish this mapping are the **cc2Xform**s that are part of each **ccPelBuffer**. See *Image Stitching tool* on page 63.

Source Image Masks

When you design your application and choose which images you will stitch together, you must decide if you need image masks. Masks are contained in separate **ccPelBuffer** objects that you create in your program. The mask images must use the same window (**ccPelRect**) as the associated source image and contain only 0 and 255 pixel values. There is a 1:1 correspondence between pixels in the image, and pixels in the mask. A 0 mask pixel value means the corresponding image pixel is ignored by the Image Stitching tool. A 255 mask value means the source image pixel is processed (stitched) by the tool. The mask allows you to be selective about which parts of an FOV are stitched into the result image. Whether you use a source mask or not is highly application dependent. Many applications will not require masks while others will benefit by having areas of a source image masked out to make the job of subsequent vision tools easier.

Image Stitching Tool Initialization

In your program setup, you must instantiate a **ccImageStitch** object to use as your Imaging Stitching tool and then call **init()** to setup the tool for your application. This tool can be reused for many stitchings, and only requires that you call its **clearImages()** method before each use to prepare the tool for a new stitching operation.

When you call **init()** you configure the tool for your specific stitching application. You pass in a vector of source windows (**ccPelRect** objects) and a vector of corresponding client transforms to the tool so it can configure and prepare its internal buffers. You also specify the output mode (*Blending* or *Overwrite)* and the pixel value the tool should use for empty pixels in the result image.

A second **init()** overload allows you to also specify the result window and its client space transform. This option is discussed in the following section.

Specifying the Result Window

The Image Stitching tool builds the result image in client coordinate space. When the stitching operation is complete, the result is available to you as a **ccPelBuffer** that includes a client space transform (**cc2Xform**) and a window into the image space (**ccPelRect**). See *Image Stitching Tool Results* on page 65.

If you do not specify a result window with **init()** as described in the previous section, the tool provides a default window which is the bounding box that encloses all of the source FOVs, and a default client transform that is identical to the source FOV1 client transform. For many applications using this default case is sufficient. If you do not want the default you must specify a result window and a result client transform when you call **init()**.

ccImageStitch provides a utility function **computeEnclosingStitchedRect()**, that computes a result window for a result client transform you specify. It computes a bounding box that completely encloses all of the source FOVs, at an orientation consistent with your result client transform. You then use this computed result window and your result client transform as inputs to **init()** to setup the tool.

If you would prefer a result window that is not a bounding box that encloses all of the source FOVs, create you own window object (**ccPelRect**) and along with your result client transform and use these as **init()** inputs to setup your tool.

Running the Image Stitching Tool

When you are ready to stitch together a group of source images, first call **clearImages()** to prepare the tool for stitching. Note: Do not call **reset()** because that will reset the tool to its default constructed state and you will have to then run **init()** again to configure the tool.

Call **addImage()** once for each source image you wish to stitch. The tool will stitch the source into the result image in client coordinate space. If your image has an associated mask use the **addImage()** overload that includes a **ccPelBuffer** for the source image and a **ccPelBuffer** for the mask. See *Source Image Masks* on page 64 for a discussion of source image masks.

Image Stitching Tool Results

After you have stitched all of the source images by successive calls to **addImage()**, the result image can be obtained by calling **stitchedImage()**. This routine returns a **ccPelBuffer** which you can then use as an input to other vision tools.

You can also call **stitchedMaskImage()** which returns a **ccPelBuffer** containing the result mask. You can use the result mask to determine which pixels in the result image were stitched from source images, and which result pixels are empty. Result masks are discussed in *Result Image Masks* on page 63.

Performance Considerations

The amount of system resources you use depends on how you use and configure the tool. The following choices will affect performance.

- More source images will cause the tool to run slower. More source images require more **addImage()** calls which increases run time.

- Using source image masks increases run time for mask processing.
- *Blending* mode requires significantly more system memory and system processing time than *Overwrite* mode.

Variable-Size Kernel Image Tools

The Variable-Size Kernel Image tools are a collection of image processing tools that work by creating an output image by applying a kernel at successive locations within an input image. Each tool uses a different mathematical operation to compute the output image pixel values.

This section and *Some Useful Definitions on page 67* give an overview of the chapter and define some terms you will encounter as you read.

Variable-Size Kernel Tools Overview on page 67 provides an overview of the tools.

How the Variable-Size Kernel Tools Work on page 69 provides detailed information about how to use the tools.

Some Useful Definitions

kernel: A 2-dimensional region which is applied to a source image.

neighborhood: The source image pixels that lie within the bounds of the kernel when applied at a given location within the source image.

source image: Image on which the image processing operation is to be performed.

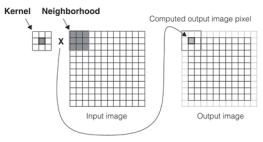
destination image: Image containing the results of the image processing operation.

mask: An image that specifies which pixels in the image or kernel to process.

Variable-Size Kernel Tools Overview

This chapter describes a collection of image processing tools that work by applying a mathematical operation to a neighborhood of pixels in an input image to produce a single pixel in an output image.

As shown in the figure below, the kernel defines a neighborhood in the input image. The kernel produces a single output value based on the values of all the pixels in the neighborhood. The tool applies the kernel at every possible location within the input image to produce the output image.



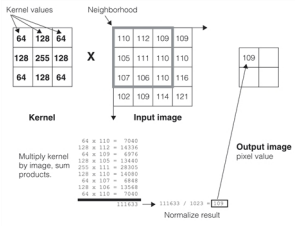
Kernel, neighborhood, and output image

What the Tools Do

The individual tools perform different operations by applying different logical and mathematical operators to compute the output image pixel values. The operation of the tools is described in this section.

Discrete Convolution Tool

The Discrete Convolution tool uses a kernel that includes an integer value for each position within the kernel. The tool works by multiplying the kernel values by the pixel values in the input image neighborhood. The products are then summed and normalized to produce the output image pixel value. The figure below shows how the discrete convolution tool computes a single pixel value in the output image.

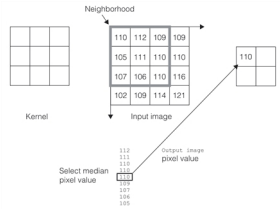


Discrete convolution

The tool normalizes the output pixel value by dividing the sum of the products by the larger of the sum of the positive kernel values or the sum of the negative kernel values.

Median Filter Tool

This tool computes the median pixel of the pixel values in the kernel neighborhood, then uses this value as the output image pixel value. Unlike the discrete convolution tool, the kernel has no values associated with it. It simply defines the shape of the neighborhood. The figure below shows how the median filter tool works.



Median filtering

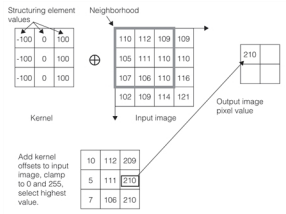
Morphology Tool

This tool performs the standard grey scale morphological operations using a structuring element that you define. The basic operations are

Erosion, where the element values are added to the input image neighborhood values and the highest of the resulting sums is used as the output value.

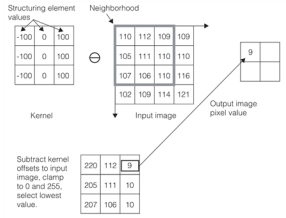
Dilation, where the element values are subtracted from the input image and the lowest of the resulting results is used as the output value.

The figure below shows an example of a 3x3 dilation.



Example of a 3x3 dilation for a single destination pixel

The figure below shows an example of a 3x3 erosion.



Example of a 3x3 erosion for a single destination pixel

The tool also supports

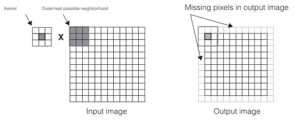
- Opening, which is an Erode followed by a Dilate
- Closing, which is a Dilate followed by an Erode

How the Variable-Size Kernel Tools Work

This section provides additional details about how these tools work.

Boundary Modes

In order for each pixel in the output image produced by a kernel tool to reflect the value of all of the pixels in the input image neighborhood, the output image will be smaller than the input image by the kernel size minus 1. This is because each pixel in the output image requires a neighborhood of pixels from the input image. When the kernel is applied to the edge of the input image, the result is a shift of the edge in the input image. This effect is shown in the figure below.



Output image size reduction

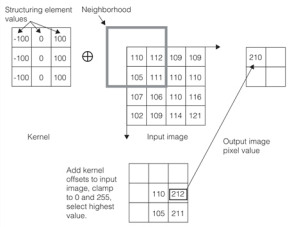
The Variable-Size Kernel Image tools provide several options for computing output image pixel values for the pixels at or near the image edge. These options allow the tool to produce an output image that is the same size as the input image.

Clipped Mode

If you specify clipped mode, then the tool kernel is only applied in locations where it is completely contained within the input image. This results in an output image that is smaller than the input image by the size of the kernel minus one. For example, if you use a 3x3 kernel, the output image is 2 pixels narrower and two pixels shorter than the input image. This mode is shown in the figure Boundary Modes in the Boundary Modes section above.

Weighted Mode

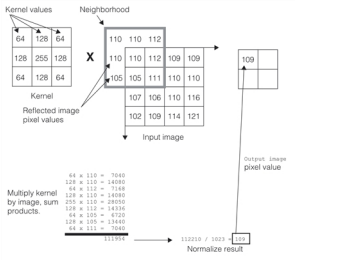
In this mode, the kernel is applied across the full image and the “missing” pixels are ignored. The output image pixel value is computed by weighting the value to reflect the number of pixel values that contributed to it.



Weighted mode

Reflected Mode

In this mode, the input image is padded with reflected pixels from the image edge so that the full neighborhood can be applied and computed.



Reflected mode

Note: The input image is not modified when you use reflected mode. Instead, the reflected pixel values are computed as required and used to compute the output image pixel value.

Tool Support for Boundary Modes

Not all tools support all boundary modes. The table below lists which tools support which modes.

Tool	Clipped	Weighted	Reflected
Median Filter	No	Yes	No
Convolve Filter	Yes	No	Yes
Morphology Filter	No	Yes	No

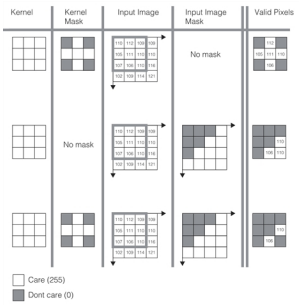
Tool support for boundary modes

Masking

The Variable-Size Kernel tools support two types of masking. You can associate a mask image with the kernel and you can associate a mask image with the input image. For both types of masks, a pixel value of 0 means ‘don’t care’ while a pixel value of 255 means ‘care’.

When the tool applies masked kernel to an input image, only pixels in the input image that correspond to care pixels in the kernel are used to compute the output image pixel value. When the tool is run using a masked input image, only pixels in the input image that correspond to care pixels in the input image mask are used to compute the output image pixel value. If a mask is supplied for both the kernel and the input image, then both the kernel pixel and the input image pixel must be care pixels for the tool to use that pixel.

The figure below summarizes the masking behavior.



Kernel and input image masks

In all cases, both kernel and input image masks must have the same size and image offset as the kernel and input image, respectively.

Tool Support for Masking

Not all tools support all mask modes. The table below lists which tools support which modes.

Tool	Kernel Mask	Input Image (Run-Time) Mask
Median Filter	Yes	Yes
Convolve Filter	No	No
Morphology Filter	Yes	Yes

Tool support for boundary modes

Median Filter Details

As described in the section *Median Filter Tool* on page 68, the median filter simply computes the median pixel value for all pixels within the neighborhood defined by the kernel. If, due to the use of either a kernel or source image mask, the median value is computed from an even number of values, the returned median is the lower of the two middle values. For example, if the neighborhood contained the values 80, 100, 120, and 140, the median value would be 100. For an odd number of values, it is of course the middle value.

When used with a 3x3 kernel, the median filter produces the same results as the existing **cfPelMedian()** CVL function.

Convolve Filter Details

As described in the section *Discrete Convolution Tool* on page 67, the convolution filter sums the products of the kernel pixels with the corresponding pixels in the input image, then normalizes the result.

The tool normalizes the result by dividing the summed products by the greater of the sum of the positive kernel values or the negative kernel values. For example, given a kernel with values -200, -100, -50, 0, 0, 100, 150, 200, and 220 the normalization divisor would be 670 (100+150+200+220).

Differences from the Fixed-Size Kernel Tool

The Variable-Size kernel tool described in this chapter differs from the fixed-size kernel discrete convolution tool described in the chapter *Discrete Convolution Tool* on page 103 in the following ways:

- The Variable-Size kernel tool does not offer a choice of normalization methods.
- The Variable-Size kernel tool does not let you set a kernel origin.

Morphology Filter Details

As described in the section *Morphology Tool* on page 68, the morphology filter computes the maximum of the kernel and image pixel sums (dilation) or the minimum of kernel and image pixel differences (erosion).

Structuring Elements

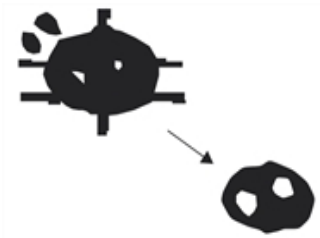
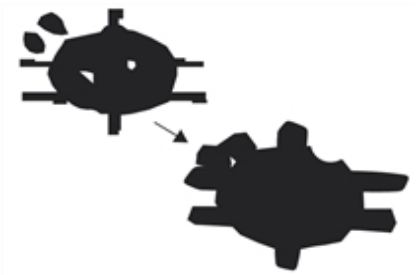
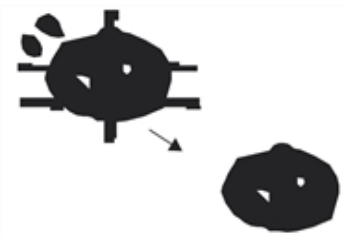
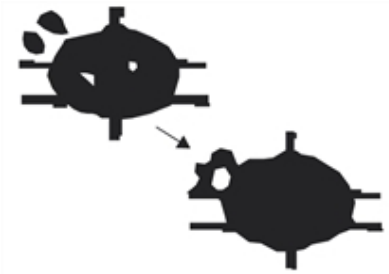
As with the discrete convolution tool, the morphology tool lets you define both a kernel, which defines the shape of the neighborhood used to compute output pixel values, and kernel values, called the structuring element.

The structuring element values are signed 8-bit integers.

By specifying a kernel mask, you can create arbitrarily shaped structuring elements.

Morphological Operations

The table below summarizes the morphological operations supported by the tool.

Operation	Example
Dilate Expand light regions, shrink dark regions.	
Erode Expand dark regions, shrink light regions.	
Close Eliminates (fills) dark gaps in light objects, smooths edges, preserves light features. A close is performed by dilating then eroding an image using the same kernel for both operations.	
Open Eliminates (fills) light gaps in light objects, smooths edges, preserves dark features. A close is performed by eroding then dilating an image using the same kernel for both operations.	

Morphological operations

Differences from Grey Morphology Tool

The Variable-Size kernel tool described in this chapter differs from the fixed-size kernel grey-scale morphology tool described in the chapter *Grey-Scale Morphology* on page 73 in the following ways:

- The Variable-Size kernel tool does not let you set an element origin.
- The Variable-Size kernel tool lets you directly specify structuring elements larger than 3x3. The fixed-size kernel tool requires that you compose 2 or more 3x3 elements to create larger elements. The Variable-Size Kernel tool includes a convenience function that lets you convert a fixed-size kernel structuring element into a variable-size kernel element.

Grey-Scale Morphology

This chapter describes the Grey-Scale Morphology tools, tools that let you apply morphological operations to the pixels in an image.

This section and *Some Useful Definitions on page 73* give an overview of the chapter and define some terms you will encounter as you read.

Grey-Scale Morphology Overview on page 73 provides an overview of grey-scale morphology.

How the Grey-Scale Morphology Tool Works on page 75 provides information that helps you make the most effective use of the Morphology tools.

Using the Grey-Scale Morphology Tool on page 78 outlines the steps to using the Morphology tools.

Some Useful Definitions

grey-scale morphological operation: Image processing operation that changes the shape of objects within an image.

morphological structuring element: Data structure that specifies a three-dimensional shape used in a morphological operation.

source image: Image on which the morphological operation is to be performed.

destination image: Image containing the results of the morphological operation.

offset value: Grey-scale value of a pixel location in a structuring element.

origin: Pixel location within a structuring element; the result of a morphological operation is stored at the pixel in the destination image corresponding to the structuring element's origin.

don't care mask: Optional mask that specifies which pixel locations in the structuring element are not to be used in the morphological operation.

dilation: Morphological operation that expands light areas and shrinks dark areas in an image.

erosion: Morphological operation that shrinks light areas and expands dark areas in an image.

opening: Erosion followed by a dilation with the same structuring element; this eliminates extraneous light details, thin lines, and small islands. It smooths object contours and maintains dark holes or narrow channels.

closing: Dilation followed by an erosion with the same structuring element; this eliminates small dark holes and gaps in light objects, and smooths edges while preserving image details.

Grey-Scale Morphology Overview

The Grey-scale Morphology tool implements morphological operations with grey-scale structuring elements on grey-scale images. Such operations are useful for filtering noise out of images or for enhancing or detecting certain image features (such as edges).

Basic Concepts of Grey-Scale Morphology

This document describes the morphological operations *dilation*, *erosion*, *opening*, and *closing*, as applied to grey-scale images. In general, dilation enlarges the size of light objects in an image, and erosion reduces their size. Opening fills gaps between close light objects in an image, and closing eliminates small light dots and thin lines. Note that in this chapter, the term *light* means "having relatively higher grey-scale values than that of background areas" and *object* and *feature* refer to areas of light pixels.

When performing a grey-scale morphological operation, you specify a *source image*, a *morphological structuring element*, and a *destination image*.

The source image is the image on which the morphological operation is to be performed. The destination image contains the result of the morphological operation. The morphological structuring element, which is used to calculate the result, is described below.

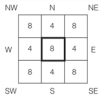
Grey-Scale Morphological Structuring Elements

A morphological structuring element is a data structure used in morphological operations. A structuring element has an *origin*, *offset values*, and a *don't care mask*. The structuring element's offset values are signed grey-scale values that are added to or subtracted from corresponding pixels in the source image. The result (the maximum of the sums or the minimum of the differences) is sent to the destination image pixel that corresponds to the structuring element's origin.

A grey-scale structuring element is implemented as a single 3x3 structuring element or as a linked list of 3x3 structuring elements. To define a grey-scale structuring element, you specify the following:

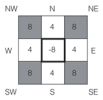
- Offset values (in the range –128 to 127) of each pixel location in the structuring element
- The origin of the structuring element (any pixel location within the structuring element)
- An optional don't care mask

The figure below shows an example of a 3x3 grey-scale morphological structuring element. Each pixel location in a 3x3 structuring element is identified by a compass position relative to the center (north, south, southeast, center, and so on) and is assigned an offset value. In this example the origin is the center pixel (outlined in bold).



3x3 grey-scale morphological structuring element

The figure below shows the same 3x3 structuring element with a don't care mask. A don't care mask specifies which pixel locations in the structuring element are not to be used in the morphological operation. In this example, the mask consists of the shaded pixels NW, NE, SW, and SE, effectively making the 3x3 square structuring element a 3x3 diamond.



3x3 structuring element with don't care mask

Dilation and Erosion

Dilation enlarges light objects in an image (regardless of whether you use positive or negative offset values). Dilation eliminates small dark gaps and holes and enlarges lighter features such as thin lines and fine detail.

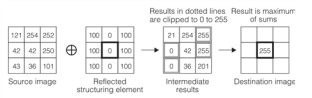
Erosion shrinks light objects in an image (regardless of whether you use positive or negative offset values). Erosion eliminates lighter features such as thin lines, fine detail, and small islands, and enlarges dark holes and gaps.

In a morphological operation, the origin pixel of the structuring element is superimposed on a pixel in the source image.

- For a *dilation* operation, the offset value at each pixel location in the structuring element is added to the value of its corresponding pixel in the source image. This yields a sum for each pixel location in the structuring element. The result is the maximum of these sums.
- For an *erosion* operation, the offset value at each pixel location in the structuring element is subtracted from the value of its corresponding pixel in the source image. This yields a difference for each pixel location in the structuring element. The result is the minimum of these differences.

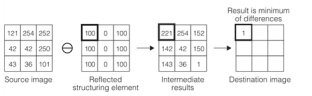
For both dilation and erosion, the result is stored at the pixel in the destination image corresponding to the structuring element's origin. The operation is repeated for each location in the source image.

The figure below shows an example of a 3x3 dilation. The bold square in the source image denotes the source pixel being operated upon; the bold square in the structuring element denotes its origin. The result of the morphological operation is written to the corresponding pixel in the destination image. (Note in the figure below that the operation uses the *reflected* structuring element; this is explained in the section *Reflection of Structuring Elements* on page 77.)



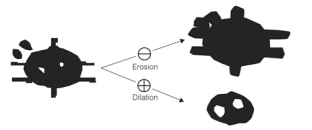
Example of a 3x3 dilation for a single destination pixel

The figure below shows an example of a 3x3 erosion.



Example of a 3x3 erosion for a single destination pixel

The figure below shows the effects of dilation and erosion using a simple 3 x 3 structuring element on an input image.



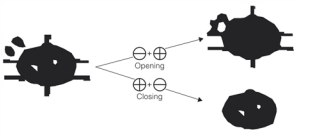
Sample erosion and dilation

Opening and Closing

Opening is performed by first eroding an image with a structuring element and then dilating the result image with the same structuring element. Opening eliminates extraneous light details, thin lines, and small islands. It maintains the relative size of objects, smooths object contours, and maintains dark holes or narrow channels.

Closing is performed by first dilating an image with a structuring element and then eroding the result image with the same structuring element. Closing fills gaps within light objects and between closely situated objects. It maintains the relative size of light objects as well as small light details, smooths object contours, and eliminates dark holes or narrow channels.

The figure below shows the effects of opening and closing on an input image.



Sample opening and closing

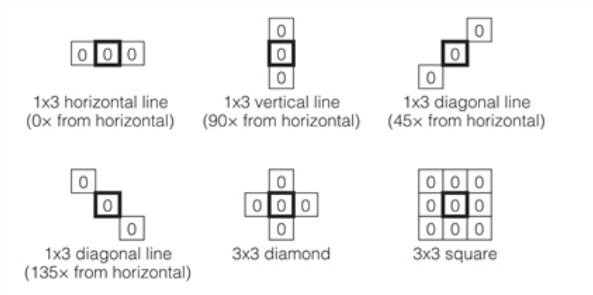
Both opening and closing tend to preserve the size and shape of large objects while affecting the size and shape of small objects.

How the Grey-Scale Morphology Tool Works

This section describes how the Grey-scale Morphology tool works.

Pre-Defined Structuring Element Library

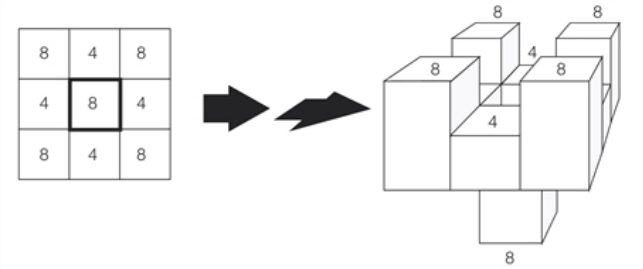
Cognex provides a standard library of six basic grey-scale morphological structuring elements that can be used individually or combined to form other structuring elements. The library structuring elements are flat; that is, all their offset values are 0. The library structuring elements are illustrated in the figure below, with their origins indicated by bold squares.



Pre-defined structuring element library

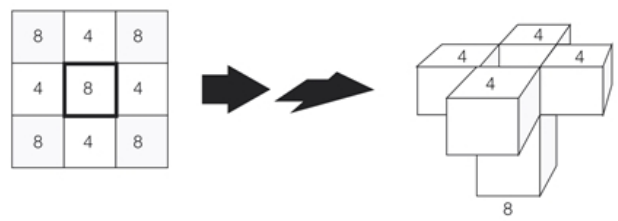
Support and Profile of Structuring Elements

It is often helpful to conceptualize the structuring element in three dimensions in order to match the shape of the structuring element to the feature you want to detect. Plotted in three-dimensional space, the structuring element lies along a horizontal plane; the structuring element's footprint on this plane is called the *support*; the offset values rise or fall perpendicular to the horizontal plane, forming the *profile*. A *flat* profile is one in which all the offset values are 0. The grey-scale profile of a 3x3 square structuring element is illustrated in the figure below.



Three-dimensional representation of a structuring element's grey-scale profile

The *don't care mask* excludes pixel locations from the morphological operation; you can think of the mask as effectively removing pixel locations from the support. The grey-scale profile of a 3x3 square structuring element with a don't care mask is illustrated in the figure below.

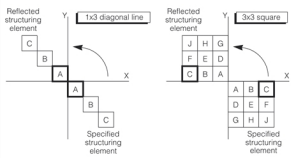


Three-dimensional representation of a structuring element's grey-scale profile (with don't care mask)

Reflection of Structuring Elements

It is important to keep in mind that the structuring element you specify is applied in reflected form in morphological operations. The Grey-scale Morphology tool automatically performs a reflection of the structuring element about the x- and y-axes. In most cases structuring elements are symmetric (such as the one shown in *3x3 grey-scale morphological structuring element* on page 74) and are unaffected by reflection about the x- and y-axes. However, if the structuring element is asymmetric, you must take its reflection into consideration.

The reflections of variously shaped structuring elements are shown in the figure below.



Reflection of structuring elements

The offset values (profile) as well as the support of the structuring element are reflected. Also, the origin is reflected.

Note: Cognex nomenclature refers to the specified origin, not the reflected origin. As illustrated in the figure above, a 1x3 structuring element whose origin is (0,0) appears to have its origin at (2,2) when reflected, and a 3x3 structuring element whose origin is (2,0) appears to have its origin at (0,2) when reflected.

Asymmetric structuring elements can be used to detect asymmetric (directional) features. For example, you must specify an upright arrow to obtain the structuring element required to detect an upside down arrow.

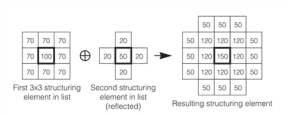
Specifying Structuring Elements Larger Than 3x3

The largest morphological structuring element you can specify directly is 3x3 pixels. To specify a larger structuring element, you must supply a list of user-defined 3x3 structuring elements that compose to the desired structuring element.

To determine the individual 3x3 elements required to compose a particular larger structuring element, you must decompose the desired structuring element. *Decomposition* is the process of calculating a list of smaller structuring elements that compose by dilation to a larger structuring element. Decomposing the larger structuring element yields a series of 3x3 structuring elements that you pass to the morphological operation. Cognex does not provide decomposition software; the decomposition calculations are your responsibility.

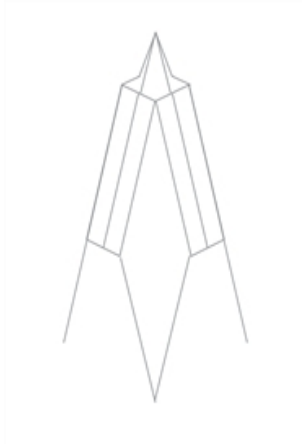
The Grey-scale Morphology tool includes a function that lets you render a graphical representation of any structuring element. By observing the rendered image, you can verify that your structuring element composes correctly, and if not, you can adjust individual 3x3 structuring elements within the list. When you have the desired structuring element, you can apply it to an image.

the figure below shows an example of structuring element composition in which a 3x3 square is dilated by a 3x3 diamond. Both are x- and y-symmetric, so the resulting structuring element is also symmetric. Note that the first 3x3 structuring element in the list is not reflected, but all subsequent 3x3 structuring elements are reflected. Another important consideration is that the dilation operation is commutative, so the order of the 3x3 structuring elements does not matter.



Composing 3x3 structuring elements by dilation

You can also obtain a three-dimensional plot of the structuring element. The figure below shows a three-dimensional representation of the 5x5 structuring element illustrated in the figure above.



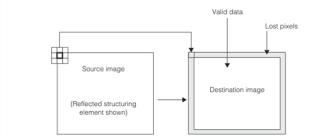
Graphical representation of a 5x5 structuring element

Output Image Size

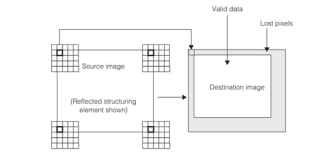
Unless the structuring element is 1x1 (that is, a single-pixel structuring element), the destination image will lose pixels from its outer borders. When a structuring element operates on pixels on the outer border of an image, some of its pixels are outside the image. Operations on such border pixels yield invalid regions in the destination image. The Grey-scale Morphology tool produces destination images that are smaller than the input images that you supply.

In a single morphological operation, the number of rows or columns of pixels lost on each side of the source image is equal to the number of rows or columns of pixels on the corresponding side of the reflected structuring element's origin (that is, not counting the origin).

For example, the invalid region for a single morphological operation (erosion or dilation) with a 3x3 structuring element whose origin is the center pixel is 1 pixel on all sides; this region is illustrated in the top figure below. The invalid region for a 4x4 structuring element whose origin is (2,2) is 1 pixel on the left edge, 2 pixels on the right edge, 1 pixel on the top edge, and 2 pixels on the bottom edge; this region is illustrated in the bottom figure below (note that the reflected origin is shown).



Lost pixels for a single operation with a 3x3 structuring element with origin at center



Lost pixels for a single operation with a 5x5 structuring element with origin at (1,1)

When you perform a compound operation such as opening or closing, you lose the above-specified amount of pixels *twice*, once for each operation. For example, the indeterminate data region for an opening or closing using a 5x5 structuring element whose origin is (1,1) would be 2 pixels on the left and top edges, and 6 pixels on the right and bottom edges.

Using the Grey-Scale Morphology Tool

For best results with the Grey-scale Morphology tool, you should observe the following general guidelines.

- Make sure that the input image contains a sufficient number of pad pixels that the loss of pixels during morphology will not destroy the features of interest in the image.
- Attempt to construct structuring elements that are the size and shape of the features in your image you are trying to accentuate or eliminate.

Labeled Projection Tool

You use the Labeled Projection tool to locate edges in circular objects, objects that are arranged radially around a center point, or objects that have an arbitrary shape.

This chapter contains the following sections:

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

annulus: An area defined by concentric inner and outer circular boundaries.

bin: One pixel of a projection image. Pixel values from the input image are added to bins within a projection image to compute a projection.

projection: Reduction of a two-dimensional array of pixels to a one-dimensional array of pixels.

projection model: An image that defines the bin into which each pixel from the input image is added to compute the projection image.

Labeled Projection Tool Overview

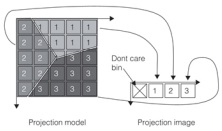
Labeled projection is the process of reducing a two-dimensional image to a one-dimensional image by assigning a *label* to each pixel in the input image, then adding all of the pixels with each label together.

Projection Models

The Labeled Projection tool projects a two-dimensional image into a one-dimensional projection image by applying a *projection model* to the input image. A projection model is an image where each pixel value indicates the bin in the projection image to which the value of the corresponding pixel in an input image is added.

The figure below shows an example of a simple projection model. The projection model defines three wedge-shaped regions within the input image. Pixels within the first region are added to the second bin in the projection region, pixels within the second region to the third bin, and pixels within the third region to the fourth bin.

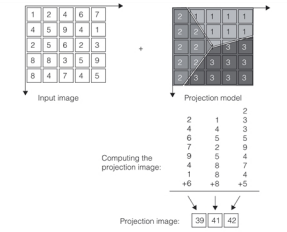
Note: The first pixel in any projection image produced by the labeled projection tool is reserved for use as a *don't care* bin. Pixels with a label of 0 are assigned to the don't care bin. There are no don't care pixels in the projection model shown in the figure below so no pixels are added to the first pixel in the projection image.



Projection Model

Projection Images

The figure below shows how the projection model shown in *Projection Model* on page 80 is applied to an input image to create a projection image. Note that in the figure below the input image has the same dimensions as the projection model.

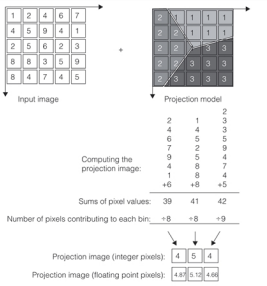


Applying a projection model

When you use the Labeled Projection tool you must supply a destination image for the projection image. The pixels in the image you supply must be large enough to hold the sum of a large number of pixel values from the input image.

You can specify either integer or floating point projection images.

The example shown in the figure above computes a non-normalized projection image: the pixel values in the projection image are the sums of all of the pixels in the input image. The Labeled projection tool also supports normalized projections, in which case the value of each bin in the projection image is divided by the number of pixels in the input image that were added to that bin. The figure below shows the same input image and projection model from the figure above, but with a normalized projection image. The figure below shows the results of supplying both an integer projection image and a floating point projection image.



Normalized projection

CAUTION: Even if you specify normalized projection, you must still provide a projection image with pixels large enough to hold the largest possible non-normalized pixel value. The Labeled Projection tool computes normalized projections by first adding all the pixel values for each label in the projection image you supply, then dividing the value by the number of pixels with each label.

Circular and Radial Projections

The labeled projection tool is often used to measure edges in radial or circular patterns. The Labeled Projection tool lets you easily create projection models that are appropriate for radial or circular patterns of edges.

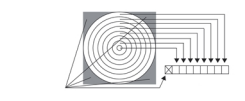
Circular Model

A *circular* projection model divides the input region into a series of concentric rings or *annuli*.

A circular projection model is divided into a series of concentric annuli. Pixels that fall within the innermost annulus are added to the second pixel in the projection image. Pixels within the outermost ring are added to the last pixel in the projection image.

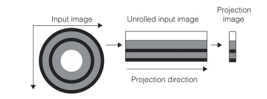
The pixels in the corners of the projection model can either be treated as don't care pixels, in which case they are added to the first pixel in the projection region, or they can be treated as part of the outermost annulus, in which case they are added to the last pixel in the projection image.

The figure below shows a circular projection model that is being used to create a projection image eight pixels in length. This projection model specifies that the corner pixels be treated as don't care pixels.



Circular projection model

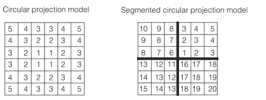
Circular projection models are useful for generating information about the edges of circular objects. The Labeled Projection tool acts as if it were unrolling the input region and creating an affine projection. The figure below shows an input image and the projection image created using a circular projection model.



Applying a circular projection model

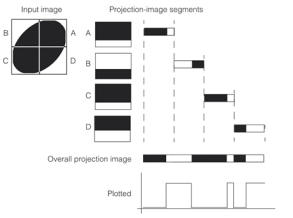
Segmented Circular Model

A *segmented circular* projection model divides the projection model into two or more equally sized sectors. Each sector contains a series of concentric arcs. The figure below compares a circular projection model and a segmented circular projection model with four segments.



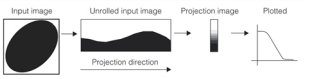
Segmented circular projection model

Segmented circular projection models are useful for generating information about the edges of elliptically-shaped objects. The Labeled Projection tool divides the input region into multiple projection-image segments which are then combined to form the overall projection image, as shown in the figure below.



Applying a segmented circular projection model

By measuring the relative spacing of the peaks in the overall projection image, you can determine the degree of eccentricity of the object. If you use an unsegmented circular projection model on the same input image generates a projection image with indistinct edge information, as shown in the figure below.

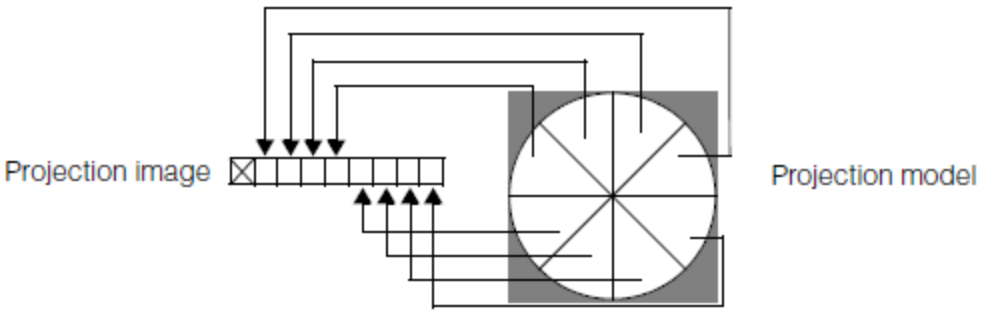


Circular projection region applied to eccentric object

Radial Model

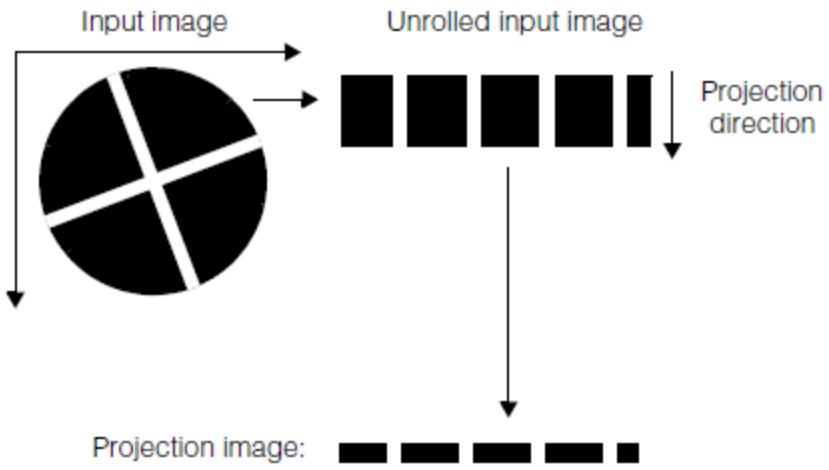
A *radial* projection model divides the projection model into a pattern of wedge-shaped sectors.

The tool sums the values of the pixels in each segment into different pixels of the projection image. As with circular projection models, you can define the corners of the image as don't care pixels, or as part of the sectors. The figure below shows a radial projection model that is being used to create a projection image eight pixels in length. The projection model shown in the figure below defines the corners as don't care pixels.



Radial projection model

Radial projection models are useful for generating information about the edges of objects arranged in a radial pattern. The Shape Projection tool acts as if it were unrolling the input region and performing an affine projection across the image. The figure below shows an input region and the projection image it generates.



Applying a radial projection model

How the Labeled Projection Tool Works

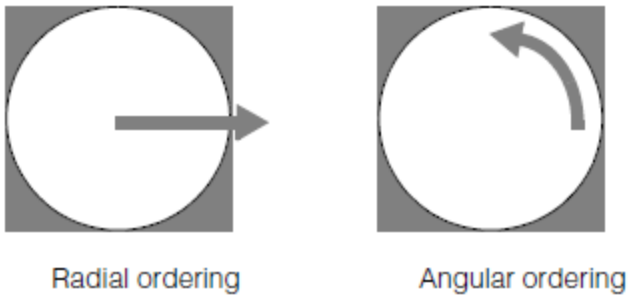
You use the Labeled Projection tool by supplying a projection model in the form of an image. You can initialize the pixel values in the image yourself, or you can use the Labeled Projection tool to construct circular and radial projection models.

Specifying Circular and Radial Models

The Labeled Projection tool lets you specify the inner and outer radii, number of sectors, the number of annuli (rings), the ordering method for regions within the projection model, and a start and stop angle. By using these parameters you can determine whether a circular or radial projection model is created. Each of these parameters is described in this section.

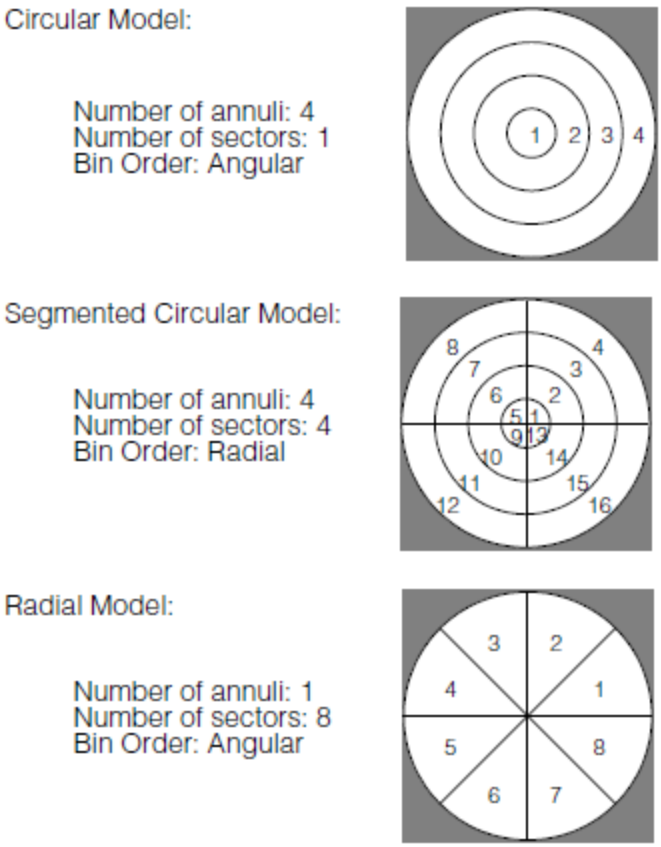
Bin Ordering Method

When you define a circular projection model, you define the number of annuli, the number of sectors, and the bin numbering scheme. The bin numbering scheme determines whether bins are numbered radially or angularly. The figure below shows the two bin numbering schemes.



Bin ordering methods

The figure below shows how you can define circular, segmented circular, and radial models by specifying the number of sectors, the number of annuli, and the bin ordering method.



Defining projection models

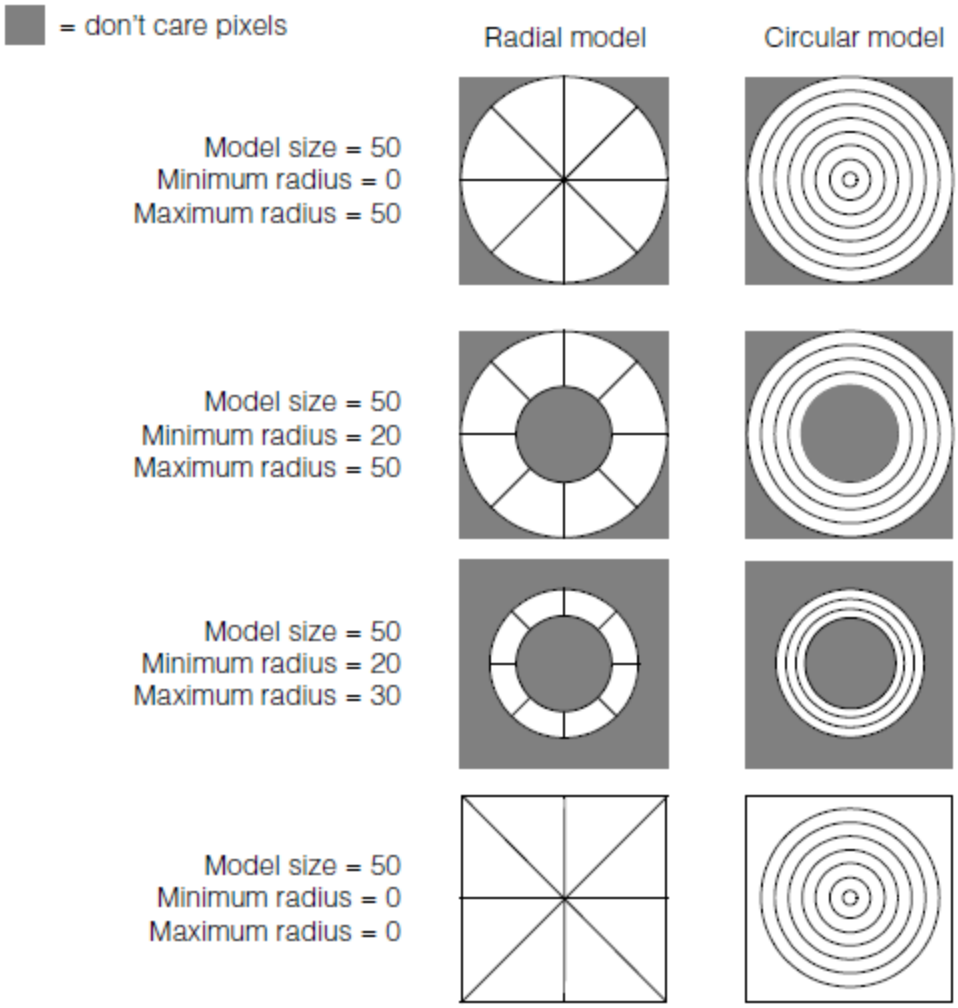
Model Size and Minimum and Maximum Radius

When you define a circular or radial projection model, you specify the overall radius of the model. The Labeled Projection tool creates and initializes a square image with a width and height equal to twice the radius you specify. By

default, the tool will set the corners of the image (all pixels within the image but outside the radius you specify) to be don't care pixels. *Circular projection model* on page 82hows the default arrangement of don't care pixels.

In addition to specifying the model image, you can also specify a minimum and maximum radius for the model. If you specify a minimum radius other than 0, all pixels within the inner radius are set to be don't care pixels. If you specify a maximum radius other than 0, all pixels outside the maximum radius are set to be don't care pixels. If you specify a maximum radius of 0, the tool sets the corner pixels to be part of the model.

The figure below shows the effect of specifying different values for the minimum and maximum radius on both circular and radial projection models. The grey box = don't care pixels, the first column is the radial model, and the second column is the circular model.

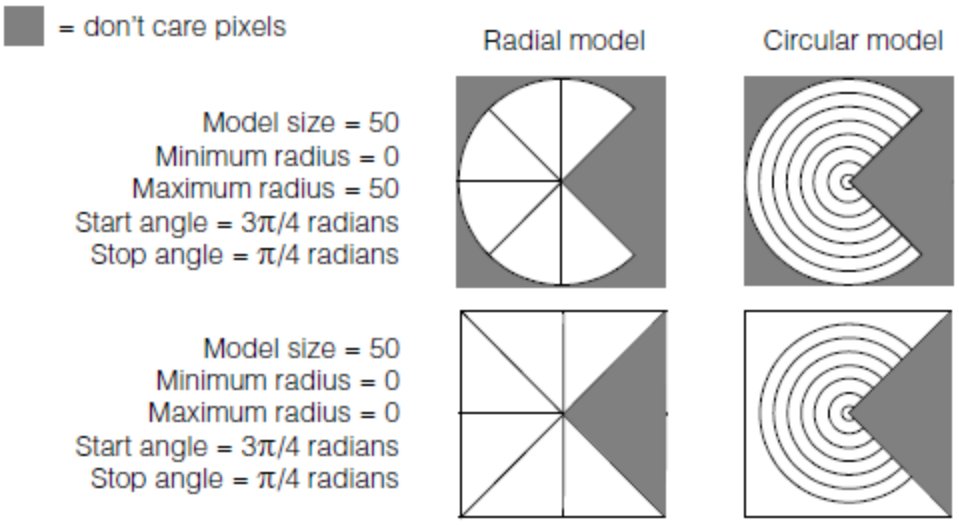


Minimum and maximum radius

Start Angle and Stop Angle

When you specify a circular or radial projection model using the Labeled Projection tool, you can specify a *start angle* and a *stop angle* for the projection model. If you specify these angles, the Labeled Projection tool will set all pixels that do not fall between the start angle and stop angle to be don't care pixels.

The start and stop angles are specified relative to the client coordinate system x-axis. The range of included angles is taken to be clockwise, starting with the start angle. The figure below shows the effect of specifying different values for the minimum and maximum radius on both circular and radial projection models.



Start and stop angle

The start angle also determines which part of the projection model is assigned to the first bin in the projection image.

Using the Labeled Projection Tool

To use the Labeled Projection tool, you following these basic steps:

1. Create a labeled projection model that corresponds to the edges of interest in the input image.
2. Apply the projection model to the input image.
3. Perform appropriate processing of the projection image.

You can also supply the projection image produced by the Labeled Projection tool as input to the Caliper tool. The Caliper tool can locate edges and peaks with sub-pixel accuracy in the projection image that you supply.

Gaussian Sampling Tool

This chapter describes the Gaussian Sampling tool, a tool that you use to smooth and sample images.

This section and *Some Useful Definitions on page 87* give an overview of the chapter and define some terms you will encounter as you read.

Gaussian Sampling Tool Overview on page 87 provides an overview of the Gaussian Sampling tool.

How The Gaussian Sampling Tool Works on page 88 provides a description of how the Gaussian Sampling tool works.

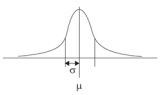
Using the Gaussian Sampling Tool on page 90 tells you how to use the Gaussian Sampling tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

Gaussian curve: A graph of the following function: $$f\left(x\right)=\frac{1}{\sigma\sqrt{2\pi}}e^{\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

where μ is the mean and σ is the standard deviation. The following figure shows a Gaussian curve:



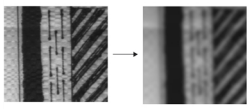
The Gaussian Sampling tool uses an approximation of a two-dimensional Gaussian curve, as shown in the following figure:

image smoothing: The process of removing or attenuating high spatial frequency information from an image

Gaussian Sampling Tool Overview

The Gaussian Sampling tool smooths an input image by applying a kernel to the image that approximates a two-dimensional Gaussian distribution. By varying the size of this kernel, you can reduce the strength of noise, or you can attenuate image features below a certain size.

The figure below shows an example of the effect of applying the Gaussian Sampling tool to an input image.



Gaussian Sampling tool applied to an input image

The application of the Gaussian Sampling tool had the affect of attenuating or eliminating much of the high-frequency information in the image. Note that most of the texture from the image in the figure above has been removed or reduced. Also, the light-colored streaks in the diagonal features on the right side of the image have been removed without changing the shape or size of the larger features.

Because few images contain meaningful information at high spatial frequencies (i.e features 1 or 2 pixels in size), the meaningful information in most images can be represented using fewer pixels than are present in the acquired image. The Gaussian Sampling tool lets you produce images that occupy fewer pixels yet still contain all of the meaningful

information about the location and shape of the features in the image. These images can be processed more quickly by other vision tools because they contain fewer pixels.

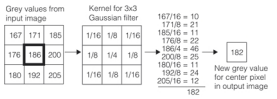
Gaussian Convolution

The Gaussian Sampling tool convolves the input image with a *Gaussian kernel* to calculate the value of each pixel in the output image. The tool examines the grey value of each pixel and the pixels surrounding it in the input image, takes a fraction of the grey value of each pixel as specified by a numerical kernel, adds these values together, and assigns this new grey value to the center pixel in the output image.

The number of pixels that the tool considers in computing the value of the pixel in the output image depends on the size of the Gaussian kernel. The tool constructs the kernel based on a smoothing value that you specify. The larger the smoothing value you specify, the larger the kernel, and the larger the features that are attenuated by the tool.

The smoothing value roughly corresponds to the feature size, in pixels, below which you wish to attenuate features. You can specify independent smoothing values in the x- and y-directions.

The figure below shows how a 3x3 pixel Gaussian kernel is applied to an input image.



Applying a 3x3 Gaussian kernel

Note: The values shown in the figure above may not be actual values that the Gaussian Sampling tool uses in a 3x3 kernel.

Smoothing and Sampling

Because of the ability of a smoothed image to represent the information in an image using fewer pixels, most applications will sample the image after smoothing. The Gaussian Sampling tool lets you combine the sampling step with the smoothing process. When the two steps are combined, they take less time to complete, improving the overall speed of most applications.

You can specify independent x- and y-axis sampling factors for the Gaussian Sampling tool.

How The Gaussian Sampling Tool Works

This section describes how the Gaussian Sampling tool works.

Specifying the Kernel Size

You specify the Gaussian kernel size by providing a *smoothness* value. The smoothness value should correspond to the feature size, in pixels, below which you wish to attenuate features.

The relationship between the smoothing value you specify and the resulting Gaussian curve's size is given by the following formula: $\sigma = \frac{\sqrt{s(s+2)}}{2}$

where s is the smoothing value and σ is the standard deviation of the resulting curve.

The size of the Gaussian kernel itself is then computed using the following formula: $w = 3s + 1$

where s is the smoothing value.

Note: You can specify independent values for smoothing in the x- and y-directions, accordingly, the kernel might not be square.

The table below gives the sigma size and kernel width for several smoothing values.

Smoothing	Sigma (s)	Kernel Width
1	.866	4
2	1.414	7
3	1.936	10
4	2.449	13
5	2.958	16

Sigma size and kernel width

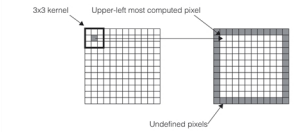
In general, you should start with a smoothing value of 1 or 2 and increase it until you obtain the desired smoothing.

Smoothing Value and Performance

The Gaussian Smoothing tool requires the same amount of time to operate regardless of the smoothing value, and hence kernel size, that you specify.

Output Image Edges

Ordinarily, whenever a kernel such as a Gaussian kernel is applied to an image, the resulting output image is smaller in size than the input image. The figure below shows the ordinary effect of applying a 3x3 kernel to an input image.



Undefined pixels resulting from ordinary kernel application

The Gaussian Sampling tool uses standard image processing techniques to compute values for the pixels at the edge of the output image, so the output image is always the same size as the input image.

In addition, if the input image is bound to a root image that has additional pixels outside the border of the input image, you can direct the Gaussian Sampling tool to use those pixels when it computes the output image.

Sampling Factor and Smoothing Value

When you apply the Gaussian Sampling tool to an input image, the tool removes or reduces all information from the image with a spatial frequency below the smoothness value that you specify. Once the image has been smoothed, the image can be sampled at any rate below the smoothing value without loss of information.

The Gaussian Sampling tool lets you specify independent sampling rates in the x- and y-direction.

Note: You should never specify a sampling rate greater than the smoothing value minus 1.

Scaling Low-Contrast Images

When you apply the Gaussian Sampling tool to input image with low contrast and low pixel values, the resulting output image can have extremely low contrast.

You can enhance the contrast of such an image by specifying an output scaling factor. When you specify an output scaling factor greater than 1.0, the Gaussian Sampling tool multiplies the value of each pixel in the input image by the

scaling factor before applying the Gaussian kernel. This has the effect of preserving information that might otherwise be lost as the kernel is applied.

You should use care in specifying an output scaling factor greater than 1.0. If the tool encounters input images with higher contrast and higher pixel values, a high output scaling factor can cause overflow.

Using the Gaussian Sampling Tool

To use the Gaussian Sampling tool, you follow these basic steps:

1. Select a smoothing value that is appropriate for your input image. For best results, start with a smoothing value of 1 and increase it until you see the desired effect.
2. Select a sampling rate. In general, the sampling rate should always be less than the smoothing value.
3. If your input images have low contrast, experiment with scaling values to improve the dynamic range of the output image.
4. Apply the Gaussian Sampling tool to a variety of input images to determine if the parameters are effective.

Histogram Tool

This chapter describes the Histogram tool, a tool that computes histograms and histogram statistics for an input image.

This section gives an overview of the chapter and define some terms you will encounter as you read.

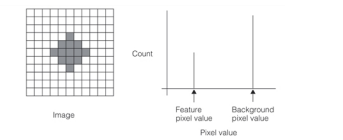
Histogram Tool Overview on page 91 provides an overview of the Histogram tool.

How the Histogram Tool Works on page 91 provides a description of how the Histogram tool works.

Histogram Tool Overview

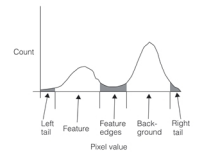
A histogram is a one-dimensional function of pixel values that represents the distribution of pixel values in a window. A histogram is represented by an array of integers where each element of the array, called a *bin,* holds a count of the number of pixels in the window with a pixel value that equals the array index of the bin. That is, the value in bin *n* is the number of pixels in the window of pixel value *n*. The total of all bin counts in the array is equal to the number of pixels in the window.

A typical histogram is characterized by the presence of *peaks*, or *modes*, representing the pixel values found in the dominant window features. As an example, consider the binary image and its histogram shown in the figure below. Only two bins have nonzero contents in this histogram: the pixel value of the feature and the pixel value of the background.



Ideal binary image and its histogram

Real images, however, seldom have histograms such as this. The effects of noise from various sources (e.g., spatial quantization error, uneven printing, irregular lighting and electrical noise) combine to spread out the peaks. A more realistic histogram of the scene in the figure above as viewed through a camera might look like the figure below. Various features of this histogram have been marked.



Pixel values in an image spread out the peaks of a histogram

Each of the peaks is clearly evident in this histogram. The number of pixels representing each peak is in the same proportion as those in the previous ideal histogram, but is now spread to involve more than two pixel values. The less populated pixel values between the two principal peaks represent the *edges* of the feature, which are neither wholly dark nor wholly light.

The left and right *tails* contain outlying points, having values that might be unreliable due to noise. In order to limit the effects of this noise, you can perform mapping on your image to ignore these unreliable end points of the histogram.

The Histogram tool lets you generate histograms from input images and compute useful statistics about a histogram.

How the Histogram Tool Works

This section describes how the Histogram tool works.

Generating a Histogram

You generate a histogram by supplying the Histogram tool with an input image. The Histogram tool returns a histogram in the form of an array of 32-bit values.

Generating Histogram Statistics

Once you have created a histogram, you can generate statistics about the histogram. The Histogram tool can compute the following statistics:

- Number of pixels
- Mean pixel value (the arithmetic average of all the pixel values)
- Median pixel value (the pixel value below which half the pixels lie)
- Modal pixel value (the most frequently occurring pixel value)
- Minimum and maximum pixel values
- Pixel value below which a supplied percentage of the pixels lie
- Standard deviation and variance of pixel values

Threshold Tool

This chapter describes the Threshold tool, a vision tool that calculates thresholds for histograms. It contains the following sections:

Some Useful Definitions on page 93 provides a glossary of relevant terms.

Threshold Tool Overview on page 93 gives an overview of the Threshold tool.

Using the Threshold Tool on page 95 provides a description of how to use the Threshold tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

bimodal histogram: A histogram that has two peaks or modes.

bin: An element in the histogram's array of integers.

histogram: A one-dimensional function of pixel values. A histogram can be represented as an array of integers. The value at element n in the array is the number of pixels in the image with pixel value equal to n.

For more information about histograms, see the chapter *Histogram Tool* on page 91.

multimodal histogram: A histogram that has multiple peaks or modes.

threshold: A value that divides a histogram into two groups of pixel values.

variance: A statistical measure of the spread of data.

within-group variance: The weighted sum of the variances for groups of pixel values.

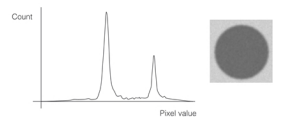
Threshold Tool Overview

The Threshold tool computes a threshold value for a histogram. You typically determine a threshold before segmenting an image with the Blob tool. For information about the Blob tool, see the chapter *Blob* on page 214.

Minimizing Within-Group Variance

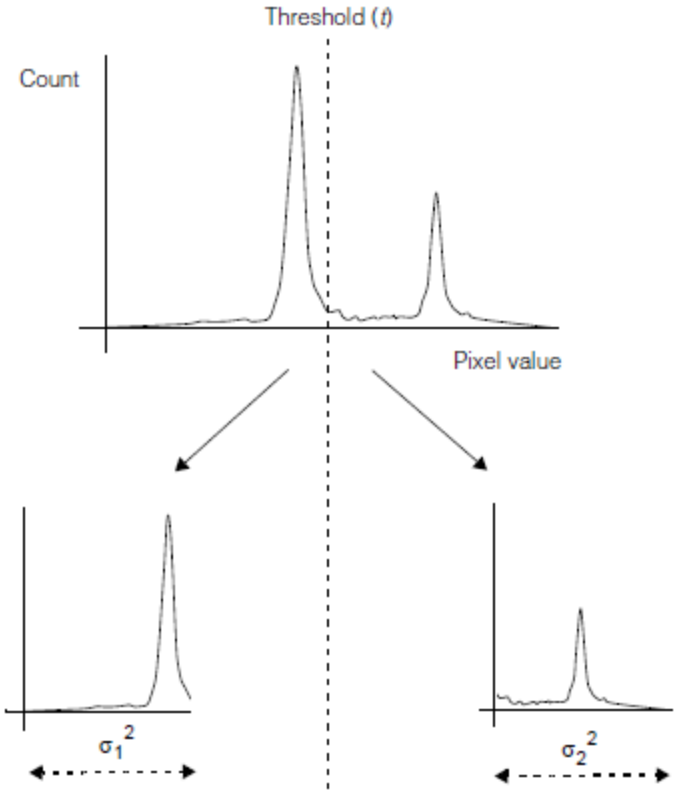
The Threshold tool uses a thresholding technique called *minimizing within-group variance* (WGV). This technique is particularly useful for obtaining a threshold value for noisy images, saturated images, or multimodal histograms.

The figure below shows an image and its associated histogram.



Object image and histogram

Any threshold *t* divides the pixel values in this histogram into two groups. As shown in the figure below, the pixel values in the resulting groups can be represented by two new histograms. σ_1^2 and σ_2^2 are the respective variances for the groups of pixel values in each of the new histograms.$\sigma_1^2\sigma_2^2$

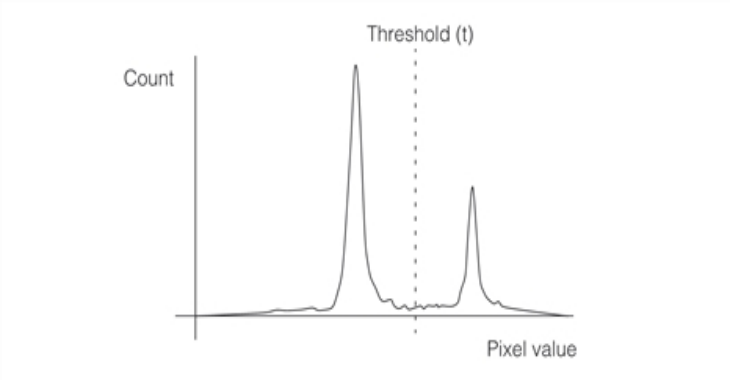


Threshold and variance of pixel values for groups

The optimum threshold divides the histogram into two groups such that each group has the minimum *within-group variance*. For any given threshold, the within-group variance $\sigma_w{}^2(t)$ is defined by the weighted sum of the variances of the two groups: $\sigma_w{}^2(t) = q_1(t)\sigma_1{}^2(t) + q_2(t)\sigma_2{}^2(t)$

where the weight factors $q_1(t)$ and $q_2(t)$ equal the number of pixels in each group divided by the total number of pixels in the image.

The Threshold tool evaluates the within-group variance for all possible threshold values, and returns the threshold that yields the minimum within-group variance. If more than one threshold minimizes the within-group variance, the tool returns the smallest value. The figure below shows the threshold returned by the tool for the image and histogram at the top of the page.



Threshold returned by the tool

Using the Threshold Tool

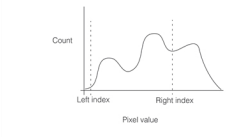
This section describes how to use the Threshold tool.

Calculating a Threshold

To calculate a threshold, call the **cfThresholdWGV()** global function, supplying the following information:

- A histogram
- A threshold result object
- The left and right indices for the partial histogram for which to calculate the threshold

The partial histogram must have at least two nonzero bins between the left and right indices. The threshold computation includes the left index value but excludes the right index value. The figure below shows how you might obtain a threshold for a portion of a multimodal histogram.



Specifying a portion of a multimodal histogram

Threshold Results

The Threshold tool reports the following result information:

- A status value that indicates whether a threshold was successfully computed
- A threshold value

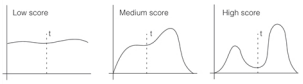
  If the Threshold tool determines multiple thresholds minimize the within-group variance, it returns the minimum threshold value.

  If the supplied histogram contains only one nonzero value, the Threshold tool cannot apply the WGV technique. In this case, the tool returns the index of the nonzero value as the threshold.

- A score for the threshold computation in the range 0.0 through 1.0.

  A greater score indicates more separation between the two groups determined by the threshold. A score of zero means that the histogram contains only one nonzero bin.

  The figure below shows the separation between groups associated with various scores.



Threshold score indicates group separation

Image Sharpness Tool

This chapter describes the Image Sharpness tool, a tool that computes the relative sharpness of an image. This chapter contains the following sections:

This section gives an overview of the chapter and defines some terms you will encounter as you read.

Image Sharpness Tool Overview on page 96 provides an overview of the Image Sharpness tool.

How the Image Sharpness Tool Works on page 98 provides a description of how the Image Sharpness tool works.

Using the Image Sharpness Tool on page 100 describes how to use the Image Sharpness tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

image sharpness: A measure of the minimum feature size present in an image

valid region: The part of an input image whose sharpness is measured

Image Sharpness Tool Overview

The Image Sharpness tool computes a measure of the relative sharpness of an input image. The primary purpose of this tool is to automate focusing a camera lens on a scene. You acquire an image and run the tool to obtain a sharpness score. You then refocus the lens changing nothing else, acquire another image and score it. By iteratively refocusing, scoring, and comparing the sharpness scores you can home in on the highest score which is the best camera focus you can obtain for your application.

Note that a single sharpness score conveys no information about the absolute sharpness of an image. Also, comparing sharpness scores of different scenes has no meaning.

Image Sharpness

Image sharpness is a measure of the degree to which an image includes the smallest resolvable features in a scene. An image's sharpness is primarily determined by the optical system used to resolve the image. The figure below shows examples of sharp and blurred images.

Sharp and blurred images

How Image Sharpness is Measured

The Image Sharpness tool provides four methods you can choose from for measuring the sharpness of an input image.

- Gradient energy
- Auto-correlation
- Band pass filtering
- Image difference at a specified offset

Each of these methods is described in the following sections.

Gradient Energy

The gradient energy of an image is computed by summing the squares of the differences between adjacent pixel values in the x- and y-directions over an entire image. The premise being that in general, a sharper image will have greater pixel-to-pixel contrast values. This technique is discussed in a paper, *Focusing Techniques*, by Subbarao, Choi, and Nizkad from the Department of Electrical Engineering, State University of New York at Stony Brook. This paper covers several focusing techniques with a mathematical proof of the soundness of each.

To calculate the gradient energy of an image you use a 3-pixel kernel as shown in the figure below.

Gradient energy kernel

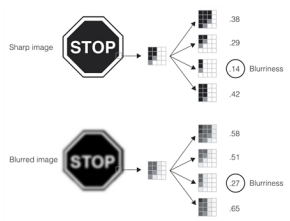
xDiff and *yDiff* are the gradient magnitude values for a given pixel location in an image. The energy gradient score for an image is the sum of the squares of *xDiff* and *yDiff* normalized by the square of the image mean grey value, summed over the entire image. The focus position that produces the highest gradient energy score produces the sharpest image.

Auto-Correlation Mode

Auto-Correlation mode works by computing the blurriness of the image, which is inversely related to the image's sharpness. Image blurriness is determined by computing the normalized correlation coefficient between an image and a slightly offset portion of the same image. The higher the correlation coefficient between two offset locations within the image, the blurrier the image is.

The Image Sharpness tool computes the normalized correlation coefficient at four offsets within the image. The smallest correlation coefficient of the four offsets is taken to be the blurriness of the image. The offsets at which the Image Sharpness tool computes the correlation coefficient are +1 and -1 pixel in both the x- and y-directions.

The figure below shows how auto-correlation mode works. For both the sharp and blurred images, a small portion of the image is shown magnified. For both images, the small image section is compared with four offset images and the correlation coefficient computed. In each case, the lowest coefficient of the four is used as a measure of the image blurriness.



Computing image blurriness in auto-correlation mode

Because the difference in pixel values across the area being compared is smaller for the blurred image, the correlation coefficients tend to be higher for the blurred image.

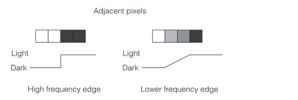
Note: In the example shown in the figure above,the actual correlation coefficients are computed for the entire image, not just the small portion shown.

The Image Sharpness tool returns a sharpness value between 0 and 1000. The sharpness value is computed using the following formula:$1000 - r \times 1000$

where r is the blurriness value (the greatest of the four computed correlation coefficients).

Band Pass Filtering Mode

Image grey scale patterns can be described in terms of frequency where sharp edges in the pixel patterns represent high frequencies, and blurred edges represent low frequencies. See the figure below.



Examples of high and low frequency edges

In this mode the tool analyzes the image looking for edges to which it assigns frequencies. You can specify a range of frequencies (band pass) where all frequencies outside this range are ignored. The sharpness score is computed from the frequencies found is the specified band.

All images contain some noise which has a very high frequency. By excluding the highest frequencies from the band pass, you can exclude noise from the sharpness calculation.

Image Difference Mode

Image difference mode computes the difference between each pixel in an image and the corresponding pixel at a specified offset within the same image. You specify an offset value that determines the offset between the pixels being compared. The greater the difference in pixel values between the sets of corresponding pixel pairs, the sharper the image.

The figure below shows how image difference mode works. A small section of both the sharp and blurred images are shown at high magnification. With an offset value of 2 pixels in the x and y direction, the difference in value for a pixel near the feature boundary is much greater for the sharp image. The overall image sharpness is measured by computing the average difference between pixel pairs at all locations within the input image.

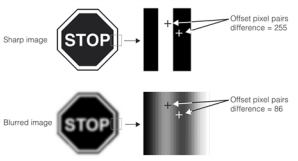


Image difference mode

Note: In the example shown in the figure above, the average difference in pixel values is computed for the entire image, not just for the magnified portion shown.

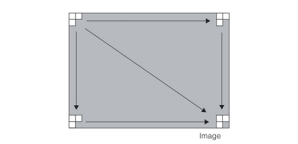
How the Image Sharpness Tool Works

The Image Sharpness tool computes image sharpness using the mode you specify.

Gradient Energy Mode

When you use the gradient energy mode you provide the Image Sharpness tool a smoothing parameter. The smoothing parameter default value is 0 which causes the tool to skip the smoothing step. If your image includes high frequency noise, you may need to apply some smoothing. To perform smoothing the tool uses the Gaussian Sampling tool. The smoothing parameter you specify is an integer number of pixels and is the input to the Gaussian Sampling tool. If smoothing is needed, generally a value of 1 will do the job, or maybe 2 on rare occasions. Too much smoothing leaves the Image Sharpness tool ineffective.

After smoothing the tool computes the image gradient energy using the 3-pixel kernel described in *Gradient energy kernel* on page 97 to compute the *xDiff* and *yDiff* gradients at each pixel position in the image as shown in the figure below.

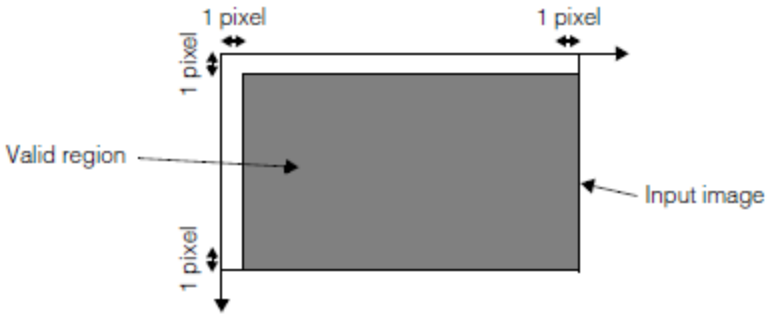


Gradient energy score calculation

Auto-Correlation Mode

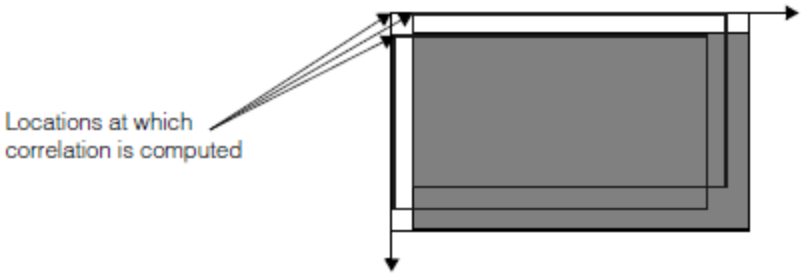
In auto-correlation mode, the Image Sharpness tool computes the normalized correlation between the image and a copy of the image offset by one pixel in the up direction, one pixel in the left direction, and one pixel in the up and left direction.

Accordingly, the valid area for auto-correlation mode is one pixel smaller in both the x- and y-directions than the input image. The figure below shows the valid region for auto-correlation mode.



Valid area for auto-correlation mode

The Image Sharpness tool computes the normalized correlation coefficient at each of the three offsets. The figure below shows the three offset locations.



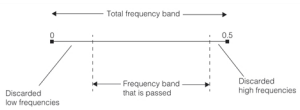
Locations at which correlation coefficient is computed

The Image Sharpness tool returns a sharpness value between 0 and 1000. The sharpness value is computed using the following formula:$1000 - r \times 1000$

where r is the blurriness value (the greatest of the four computed correlation coefficients).

Band Pass Filtering Mode

When you specify the *eBandPassFiltering* mode in the sharpness parameters you also specify a band pass range. The entire range of possible frequencies the tool can find in an image is in the range 0 through 0.5. You should use your knowledge of the image content to restrict the range to the frequencies you expect. By doing this you can cause the tool to discard (filter out) unwanted frequencies that may be present, such as high frequency noise and low frequency image background. See the figure below.

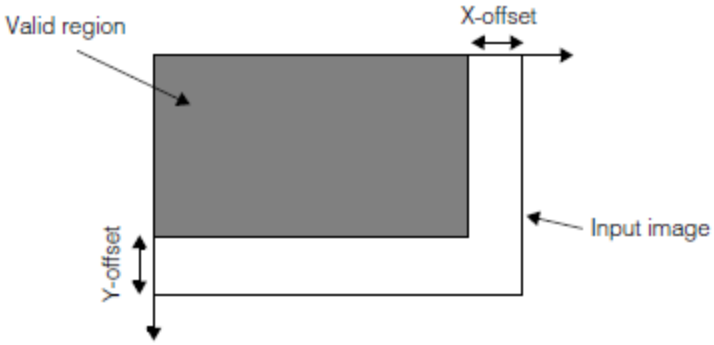


eBandPassFiltering mode

Image Difference Mode

In image difference mode, the Image Sharpness tool performs a pixel-by-pixel comparison between the image you supply and the same image offset by the x- and y-axis offset values that you specify.

The portion of the image that is actually compared is determined by the image offset values that you specify. Only the *valid region* of the image is considered when computing the sharpness value in image difference mode. The figure below shows how the valid region is computed for image difference mode.



Valid area for image difference mode

The image difference value that is returned is normalized by the number of pixel-pairs compared. The greatest possible value that can be returned is equal to the maximum pixel value for the input image (255 for an 8-bit image).

Using the Image Sharpness Tool

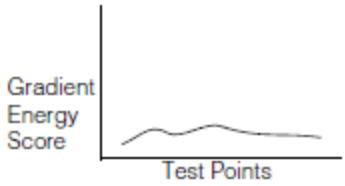
This section contains suggestions for using the Image Sharpness tool in your application.

Selecting the Mode

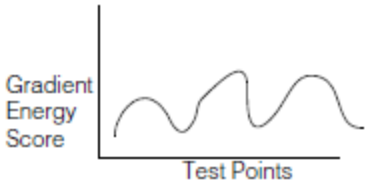
This section offers some guidelines for choosing the sharpness tool mode. These guidelines are based on experiments done by Cognex. We suggest you start with these recommendations and if you are not achieving good results, try your own experiments to find the mode that works best on your images.

Try the modes in this order:

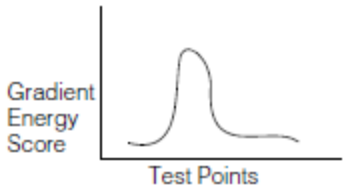
1. Gradient Energy mode
    - Works best for most images.
    - Start with no smoothing and add smoothing only if necessary. Use the following diagrams as a guide to whether to add smoothing.



Too Much Smoothing



Too Little Smoothing



Correct Smoothing

2. Auto-Correlation mode
    - Works well with many images.
    - Good for low contrast images.
3. Band Pass Filtering mode
    - Good for very low contrast images where the difference between features and background can be as few as 10 pixels.
    - Good for images where the background is textured.
    - Good for images with a high noise content.
4. Image Difference mode
    - Requires that you know the size of the features of interest in the image. If you specify an offset that is different from the feature size, the tool may not return reliable results.
    - Because image difference mode measures image difference in the x- and y-directions, it cannot determine the sharpness of diagonal features.
    - Image difference mode may not work well on high-frequency scenes.

Performance Considerations

This section describes some of the factors that affect the amount of time required by the tool to compute a sharpness score.

1. In image difference mode, the amount of time required to compute a sharpness score for a particular image is affected by the following factors:

- The size of the valid region (larger regions require more time)
- The size of the image offsets (smaller offsets require more time)

2. In auto-correlation mode, the amount of time required to compute a sharpness score is affected by the size of the input image (larger images require more time).
3. In general, for large image offsets, image difference mode is faster than auto-correlation mode for a given image size. If, however, you need to determine the sharpness of an image with mostly small features, auto-correlation mode may be faster.
4. For most images, band pass filtering mode is a slower procedure than other modes.
5. In gradient energy mode if you specify a *lowPassSmoothing* parameter value greater than zero, the tool will run slower.

Discrete Convolution Tool

This chapter describes the Discrete Convolution tool, a tool lets you convolve images with an arbitrary 3x3 kernel.

This section and *Some Useful Definitions on page 103* give an overview of the chapter and define some terms you will encounter as you read.

Discrete Convolution Tool Overview on page 103 provides an overview of how the Discrete Convolution tool works and how you use it.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

convolution: The creation of a new image where each pixel value is the sum of the products of an input image's pixel values by those of a kernel.

energy: A measure of the overall intensity of an image.

kernel: A small, square image used to perform image processing operations.

Discrete Convolution Tool Overview

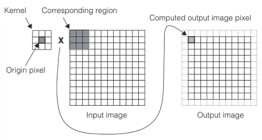
The Discrete Convolution tool lets you convolve an image with an arbitrary 3x3 kernel.

Image Convolution

The Discrete Convolution tool works by convolving the input image with a 3x3 kernel to calculate the value of each pixel in an output image. The tool computes a new pixel value for each possible location within the input image.

For each location, the tool multiplies the grey value of each of the nine pixels in that region of the input image by the values of the corresponding pixels in the kernel. It then adds these values together and assigns this new value to the pixel in the output image that corresponds to the origin of the kernel.

The figure below shows how the Discrete Convolution tool computes the first pixel in the output image. Note that since the kernel requires 9 input pixels to produce a single output pixel and the kernel is always fully contained within the input image, the resulting output image is smaller than the input image.

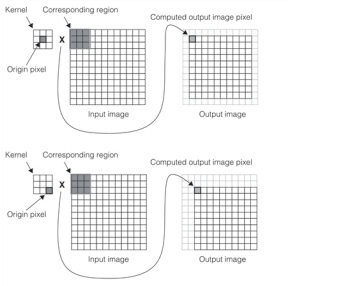


Convolving an image with a kernel

The Discrete Convolution tool performs the operation shown in the figure above at every possible kernel location within the input image.

Kernel Origin

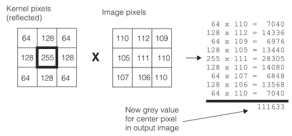
When you create a kernel for use with the Discrete Convolution tool, you must specify the origin pixel of the kernel. The tool places the output pixel at the location in the output image that corresponds to the location of the kernel origin. The figure below shows the effect of specifying different kernel origins on the convolution shown in *Convolving an image with a kernel* on page 103.



Kernel origin affects offset but not size of output image

Computing Output Pixel Values

The Discrete Convolution tool supports only images and kernels comprising pixels with integer values. The figure below shows the effect of computing an output pixel value using an 8-bit kernel and 8-bit input image.



Computing the output pixel value

Note: The kernel is reflected before the values are multiplied. Since the kernel shown in the figure above is symmetrical, the reflection has no effect.

The value produced by simply multiplying the kernel pixel values by the input image pixel values then summing the results is larger than the largest value that can be stored as a pixel value.

In order to create usable output images *and* compute output images with maximum accuracy, the Discrete Convolution tool's default behavior is to normalize the output pixel values by automatically prescaling the kernel pixel values and normalizing the output pixel values. You can override this behavior and specify your own normalization factor.

Automatic Kernel Prescaling and Normalization

If you specify automatic kernel prescaling and normalization (the tool's default), the Discrete Convolution tool multiplies all of the kernel pixel values by a prescaling factor equal to $\frac{32767.0}{max(\Sigma pos, \Sigma neg)}$

where

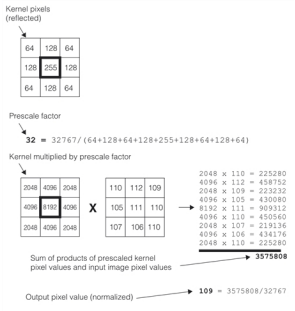
Σpos is the sum of the kernel pixel values greater than 0.

Σneg is the absolute value of the sum of the kernel pixel values less than 0.

The sum of the result of multiplying the prescaled kernel values times the input image pixel values is normalized by dividing it by 32767 if the output image is an 8-bit image.

Note: The normalization is performed by right-shifting the output value by 15 bits.

The figure below shows the effect of automatic kernel prescaling and normalization on the example kernel and input image shown in *Computing the output pixel value* on page 104.



Kernel prescaling and normalization

The automatic kernel prescaling and normalization method supported by the Discrete Convolution tool will never experience overflow, and when used with kernels with positive pixel values, it will always preserve the input image's energy level.

If you are using 16-bit input and output images, automatic prescaling and normalization is done using the same method, except that the output values are right-shifted by only 8 bits (divided by 256). This preserves as much precision as possible for use in intermediate computations. To preserve the energy value of the original input image, you should right-shift the final output values by an additional 7 bits (divide by 128).

Specifying a Normalization Factor

You can override the automatic kernel prescaling and normalization of the Discrete Convolution tool by prescaling the kernel yourself (using 16-bit kernel pixels) and then supplying a normalization factor to the Discrete Convolution tool. You specify the normalization factor as the number of bits to right-shift the output pixel values.

Sample Convolve Tool

This chapter describes the Sample Convolve tool, a tool lets you perform a simultaneous separable convolution and sampling on images.

This section and *Some Useful Definitions on page 106* give an overview of the chapter and define some terms you will encounter as you read.

Sample Convolve Tool Overview on page 106 provides an overview of how the Sample Convolve tool tool works and how you use it.

Some Useful Definitions

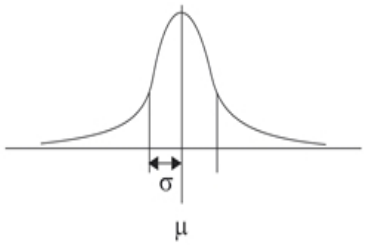
This section defines some terms and concepts used in this chapter.

convolution: The creation of a new image where each pixel value is the sum of the products of an input image's pixel values by those of a kernel.

kernel: A small, rectangular image used to perform image processing operations.

Gaussian curve: A graph of the following function: $f\left(x\right) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$

where μ is the mean and σ (sigma) is the standard deviation. The following figure shows a Gaussian curve:



The Sample Convolve tool uses an approximation of a two-dimensional Gaussian curve, as shown in the following figure:

image smoothing: The process of removing or attenuating high spatial frequency information from an image

Sample Convolve Tool Overview

The Sample Convolve tool lets you perform simultaneous separable convolution and sampling on images. Separable means that the 2D kernel used during convolution can be separated into two 1D kernels (x- and y-directions); that is, the vector product of the two 1D kernels yield the 2D kernel. This means that the 2D convolution performed with the 2D kernel yields the same result as the two convolutions performed one after the other using the 1D kernels.

The Sample Convolve tool is similar to the *Gaussian Sampling Tool on page 87*. A common use of the Sample Convolve tool is downsampling with Gaussian smoothing. For a description of convolution with an example, see the section *Gaussian Convolution* on page 88.

Note that standard downsampling is done with sigma = 0.5 * **floor(***sample***)**, where *sample* is the downsampling factor. This is a good compromise of speed, avoiding the introduction of artifacts, and maintaining the information content. For example, downsampling by a factor of 3.5 (*sample* = 3.5) would use a sigma of 1.5. When creating a Gaussian kernel using the provided functions, the kernel is defined in the range of -2*sigma to 2*sigma.

Standard upsampling uses no smoothing.

Advantage of the Sample Convolve Tool over the Gaussian Sampling Tool

When performing standard downsampling with a downsampling factor greater than or equal to 2, the Sample Convolve tool is significantly faster than the Gaussian Sampling tool. If you perform smoothing without downsampling, the Gaussian Sampling tool may perform as good as the Sample Convolve tool or better, however, Cognex recommends that you use Sample Convolve tool in all cases.

How the Sample Convolve Tool Works

This tool operates in image space, and all input parameters are in image units. Most interfaces use source image units, but destination image units are used where you specify so.

Client coordinates are respected, that is, the client coordinates in the destination image correspond to the client coordinates in the source image.

The tool creates a destination pel for each place where a sample falls inside the source image window. The left and top boundaries of the image are considered in the image; the right and bottom boundaries are not.

To use the tool, perform the following steps:

1. Set the Sample Convolve parameters including the 2D kernel using a **ccSampleConvolveParams** object. Note that you can specify either the same or different kernel sizes and sampling values in the x- and y-directions.

   In addition, if the source image is bound to a root image that has additional pixels outside the border of the source image, you can direct the Sample Convolve tool to use those pixels when it computes the destination image.
2. Run the **cfSampleConvolve()** global function by supplying a source image and the **ccSampleConvolveParams** object to it, and providing a destination image to contain the result image of the convolution.

In addition to setting the Sample Convolve parameters, you can use the **ccSampleConvolveParams** object to compute the following:

- Maximum destination region that could result from the given source region and parameters.
- Maximum source region that would be referenced by the convolution.

Specifying the 2D Kernel

You can specify the 2D kernel in the following two ways. If you specified the kernel in one way and you want to specify another kernel in the other way, you can easily do this, the kernel stored in the tool will be overwritten by the new kernel.

Specifying the 2D Kernel Based on a Sigma or Sample Value

You can have the Sample Convolve tool compute a 2D Gaussian kernel based on a sigma or sample value you provide. As mentioned previously, if you provide a sample value, sigma will be calculated as follows:

sigma = 0.5 * **floor(***sample***)**

To set up the 2D kernel based on the sample value, use **ccSampleConvolveParams::setGaussSample()**.

To set up the 2D kernel based on the sigma value, use **ccSampleConvolveParams::setGaussSmoothing()**. The sample rate is left unchanged if you use this function. If you want to change the sample rate, and as a result not work based on the above formula between sample and sigma, use the **ccSampleConvolveParams::sample()** function.

Specifying the 2D Kernel Based on Two 1D Kernels

You can specify two 1D kernels, one for the x-direction and one for y-direction; the Sample Convolve tool calculates the 2D kernel as the vector product of the 1D kernels.

For example, if kernelX = [kx1, kx2] and kernelY = [ky1, ky2, ky3], then the 2D kernel is the following:

$$kernel2D = kernelY^T \times kernelX = \begin{bmatrix} ky1 \\ ky2 \\ ky3 \end{bmatrix} \times \begin{bmatrix} kx1 & kx2 \end{bmatrix} =$$

kx1 x *ky1*	*kx2* x *ky1*
kx1 x *ky2*	*kx2* x *ky2*
kx1 x *ky3*	*kx2* x *ky3*

Note: Because the 2D Gaussian curve function (depending on x and y) is x-y separable, you can create a Gaussian 2D kernel by specifying two 1D Gaussian kernels as well.

To set up the 1D kernel in the x-direction, use the **ccSampleConvolveParams::kernelX()** function. To set up the 1D kernel in the y-direction, use the **ccSampleConvolveParams::kernelY()** function. Also, set the sampling value using the **ccSampleConvolveParams::sample()** function.

Image Transformation Tools

This chapter describes the following image transformation tools:

- The Affine Rectangle Sampling tool, a vision tool that lets you create images by sampling an input image within an affine rectangle
- The Projection tool, a vision tool that generates a one-dimensional projection image from an affine rectangle within a two-dimensional image by summing columns of pixels within the affine rectangle
- The Polar Coordinate Transformation tool, a vision tool that transforms Cartesian coordinates to polar coordinates.

You can use the Affine Rectangle Sampling tool to rotate input images so that they are aligned with an image's image coordinate system. You can use the Projection tool to isolate and accentuate edges within images. You can use the Polar Coordinate Transformation tool, for example, to unwrap images.

This section and *Some Useful Definitions on page 109* give an overview of the chapter and define some terms you will encounter as you read.

Affine Rectangle Sampling Tool on page 110 provides an overview of affine rectangles and image sampling and describes how the Affine Rectangle Sampling tool works.

Projection Tool on page 118 provides an overview of affine projection and describes how the Projection tool works.

Framework Differences on page 120 summarizes the differences between the OMI and CVL versions of the Affine Rectangle Sampling and Projection tools.

Polar Coordinate Transformation Tool on page 121 defines the polar coordinate system and its relationship to the more familiar Cartesian coordinate system, describes how the Polar Coordinate Transformation tool works, and how to use the tool to its best advantage.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

affine rectangle: A quadrilateral where the opposite sides are parallel to each other

Cartesian image: An image in which the pixels are square and in which the coordinates of a point are represented as x and y.

height: Height of an image. For polar conversion, the height is the radial dimension (in pixels) of the polar destination image.

interpolation: Method of estimating the grey-level value of a sub-pixel location in an image based on the pixel values surrounding the location

polar image: An image in which the pixels look like wedges or keystones.

projection: Reduction of a two-dimensional array of pixels to a one-dimensional array of pixels. A projection is formed by summing the pixel values along a ray within the projection region.

ray: Line drawn in the direction of a projection along which pixel values are summed

rotation: The turning of an object or rectangle about an axis point or center.

rotation signature: A polar projection of an object. By default, the polar radius is stored in the y dimension, polar angle in the x dimension.

skew: The distortion of a rectangle into a parallelogram

translation: The movement of an object in x-y space.

width: Width of an image. For polar conversion, the width is the circumferential dimension (in pixels) of the polar destination image.

Affine Rectangle Sampling Tool

The Affine Rectangle Sampling tool produces an image by sampling the contents of an affine rectangle within an input image. A common application for the Affine Rectangle Sampling tool is to correct for scene rotation before applying non-rotationally-invariant vision tools to an image.

Affine Rectangle Sampling Tool Overview

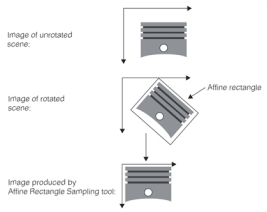
This section provides an overview of the Affine Rectangle Sampling Tool.

Aligning Rotated Objects

If your application uses vision tools which are not rotation-invariant, such as CNLSearch, you may need to correct for *scene rotation* before applying vision tools to input images.

For example, if you are using a non-rotation-invariant tool to measure a part, and input images may contain rotated instances of the part, you can use the Affine Rectangle Sampling tool to produce an un-rotated version of the input image.

The figure below shows an example of using the Affine Rectangle Sampling tool to correct for a rotated input image.

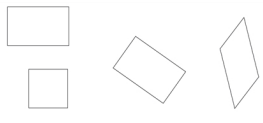


Using affine rectangle sampling to correct scene rotation

You can also use the Affine Rectangle Sampling tool to create images from skewed or scaled images of scenes.

Specifying an Affine Rectangle

An affine rectangle is any quadrilateral where the opposite sides are parallel to each other. Squares, rectangles, and parallelograms are all affine rectangles. The figure below shows examples of affine rectangles.

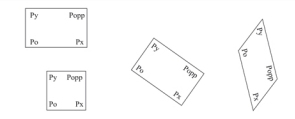


Affine rectangles

The affine rectangles used by the Affine Rectangle Sampling tool and the Projection tool have labeled vertices. The four vertices are:

- Origin point (shown as Po)
- X vertex point (shown as Px)
- Y vertex point (shown as Py)
- Opposite vertex point (shown as Popp)

The figure below shows each of the affine rectangles from the figure above with their vertices labeled.

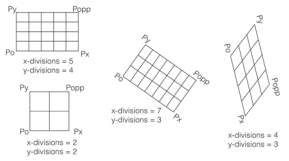


Labeled affine rectangles

Sampling the Affine Rectangle

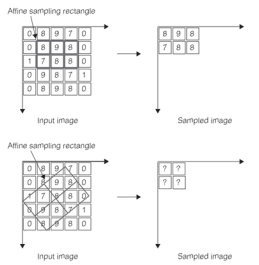
The definition for sampling an affine rectangle within an image includes both the definition of an affine rectangle and the number of *affine pixels* into which it is divided. The number of affine pixels is determined by the number of divisions along the Po-Px and Po-Py axes applied to the rectangle.

The figure below shows each of the affine rectangles shown in *Affine rectangles* on page 110, with the number of x-axis and y-axis divisions.



Affine rectangles and affine pixels

Affine rectangles which represent aligned orthogonal transformations of the input image client coordinate system can be used to create affine samples without the need to perform complicated processing. If the affine rectangle does not, some kind of sampling must be applied to produce a sampled image. The figure below illustrates this problem.



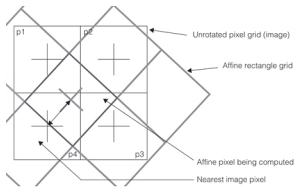
Orthogonal and non-orthogonal affine sampling rectangles

The problem of determining the pixel values when sampling an affine rectangle that is not aligned to the pixel grid of the input image is solved by *sampling* the pixel grid to produce the sampled image. The Affine Rectangle Sampling tool supports three types of sampling: *uninterpolated sampling*, *bilinear interpolation*, and *high-precision interpolation*.

Uninterpolated Sampling

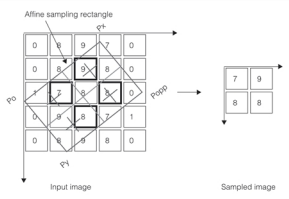
Uninterpolated sampling computes the value for each pixel in the sampled image by determining the pixel value of the pixel whose center is closest to the center of the affine pixel. This is called *nearest neighbor* sampling.

The figure below shows how uninterpolated sampling works.



Uninterpolated sampling

The figure below shows how uninterpolated sampling would produce the affine sampled image shown in *Orthogonal and non-orthogonal affine sampling rectangles* on page 111. The centers of the affine pixels are shown as grey crosses, and the nearest pixels in the input image are shown with heavy borders.

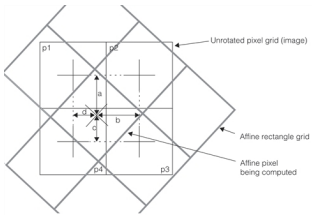


Computing a non-interpolated sample image

Bilinear Interpolation

Bilinear interpolation considers the values of the *four* pixels closest to the center of each affine pixel in the affine sampling rectangle. The distance-weighted average of the values of the four pixels is computed and used for the value of the affine pixel.

The figure below shows how bilinear interpolation is computed.



Bilinear interpolation

The value of the interpolated affine pixel is computed using the following formula:

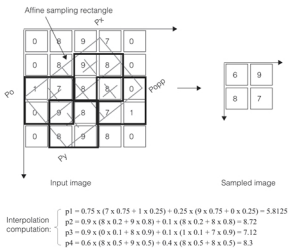
$$(1-a)\,(p2\times(1-b)+p1\times b)+a\,(p3\times(1-b)+p4\times b)$$

where

a, b, c, and d are the four distances shown in the figure above, normalized so that the distance between pixel centers is 1.

p1, p2, p3, and p4 are the values of the four pixels shown in the figure above.

The figure below shows how bilinear interpolation would produce the affine sampled image shown in *Orthogonal and non-orthogonal affine sampling rectangles* on page 111. The centers of the affine pixels are shown as grey crosses, and the four pixels used to compute each affine pixel are surrounded by a heavy border.



Computing a interpolated sample image

The computed interpolated pixel values are rounded to the nearest whole integer value.

Note: Depending on which platform you are using, the interpolation arithmetic might be done using scaled integer math or floating point math. In general, this has no effect on the resulting image.

High-Precision Interpolation

High-precision interpolation is similar to bilinear interpolation except that it considers additional pixels in determining the value of a sampled pixel.

In general, high-precision interpolation offers higher accuracy interpolation than does bilinear interpolation at a cost of slower execution speed.

How the Affine Rectangle Sampling Tool Works

This section describes how the Affine Rectangle Sampling tool works.

Specifying Affine Rectangles

There are three ways you can specify an affine rectangle using the Affine Rectangle Sampling tool.

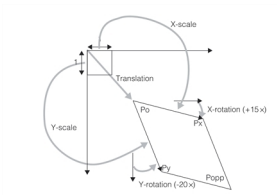
- A transformed unit square
- A set of three vertices
- The rectangle's location, size, and angles of rotation and skew

Each of these methods is described in this section.

Specifying a Transformed Unit Square

If the programming interface you are using supports transformation objects, you can specify an affine rectangle by supplying the transformation that transforms a *unit square* into the affine rectangle of interest, where the unit square is a 1 x 1 rectangle whose origin is at the origin of the client coordinate system.

The figure below shows an example of a transformed unit square.

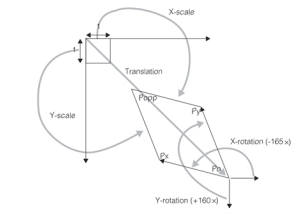


Affine rectangle defined by transformed unit square

Each of the individual components of the transformation is illustrated in the figure above as a grey arrow.

Note that the x-rotation and y-rotation values determine the locations of the Po origin point. The x-rotation and y-rotation angles are the angles from the coordinate system x and y axes to the Po-Px and Po-Py sides of the affine rectangle.

The figure below shows the effect of changing the x-rotation, y-rotation, and translation values that define the affine rectangle shown in the figure above. While the location of the rectangle is the same as it is in the figure above, the locations of the vertices are different.

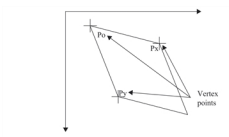


Changing vertices' locations by changing x rotation, y rotation, and translation

Specifying Three Vertices

Since each affine rectangle is defined by three of its four labelled vertices (Po, Px, and Py), you can completely specify an affine rectangle by supplying the client coordinates of each of those vertices.

The figure below illustrates the same affine rectangle shown in *Affine rectangle defined by transformed unit square* on page 113 defined by the locations of its three vertices.

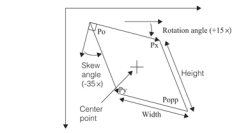


Affine rectangle specified by three vertex points

Specifying Location, Size, and Angle

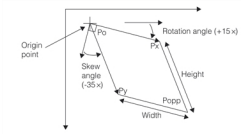
You can specify an affine rectangle by supplying the location of its center point, its width (defined as the distance between Po and Px), its height (defined as the distance between Po and Py), its angle of rotation (defined as the angle between the client coordinate system x-axis and the side of the rectangle defined by Po and Px), and its angle of skew (defined as the angle between the side of the rectangle defined by Po and Py and a line drawn perpendicular to the side of the rectangle define by Po and Px).

The figure below illustrates the same affine rectangle shown in *Affine rectangle defined by transformed unit square* on page 113 defined by the location of its center point, its size, and its angles of skew and rotation.



Affine rectangle specified by center point, size, and angles of skew and rotation

You can also specify an affine rectangle by the location of the origin point. The figure below illustrates the same affine rectangle shown in *Affine rectangle defined by transformed unit square* on page 113 defined by its origin, size, and angles of skew and rotation.



Affine rectangle specified by origin, size, and angles of skew and rotation

Specifying Affine Sampling Parameters

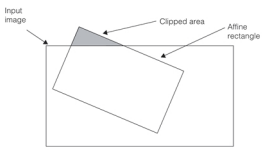
Beyond the specification of the affine rectangle, only three additional items of information are required to specify the affine sampling parameters.

- The number of x-axis divisions
- The number of y-axis divisions
- The sampling method

Each of these items is described in the section *Affine Rectangle Sampling Tool Overview* on page 110.

Clipping

When you actually apply an affine rectangle and affine rectangle sampling parameters to an input image, the specified affine rectangle may extend outside the bounds of the input image. This is called *clipping*. The figure below illustrates clipping.

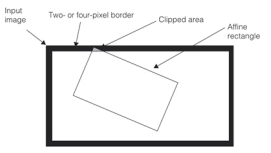


Clipping.

Because of the way the Affine Rectangle Sampling tool is implemented, clipping will occur if the center of any affine pixel within the affine rectangle is within two pixels of the edge of the image being sampled. Note that the size of the affine pixels you define (which is controlled by the number of x-axis divisions and y-axis divisions you specify) affects whether or not an image is clipped.

Note: If you specify high-precision interpolation, as described in the section *High-Precision Interpolation* on page 112, clipping will occur if the center of any pixel is within *four* pixels of the edge of the image being sampled.

The figure below shows the effect of this boundary on clipping. Even though the affine rectangle is completely enclosed within the input image, clipping still occurs because the center of an affine pixel is too close to the edge of the input image.



Clipping.

You can call a function to determine if a particular set of affine sampling parameters will be clipped by a particular image without needing to perform the affine transformation itself.

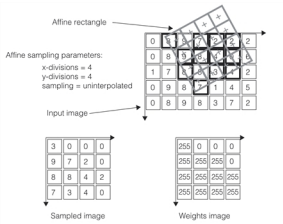
Controlling How Clipping is Handled

You have two options for controlling how the Affine Rectangle Sampling tool handles clipping. You can configure the tool to throw an error if clipping occurs, or you can configure the tool to return a *weights image* if clipping occurs.

A weights image is an image with the same dimensions as the sampled image. The value of each pixel in the weights image indicates whether the value of that pixel was affected by clipping.

Pixels in the sampled image that were computed using all input pixels called for by the specified interpolation method are assigned the weight 255. If one or more of the input pixels is not available because of clipping, the weight is set to a fractional value between 0 and 255 that represents the fraction of input pixels available. By examining the weights image pixels, you can determine the effect of the clipping. For example, a weight of 127 indicates that only half of the input pixels were available for computing the sampled image pixel, and a weight of 0 indicates that none of the input pixels were available.

The figure below shows an example of an input image, an affine rectangle with sampling parameters, and the resulting sampled and weights images.



Computing sampled and weights images

Sampling Methods

When you choose between uninterpolated sampling, bilinear interpolation, and high-precision interpolation, keep the following in mind:

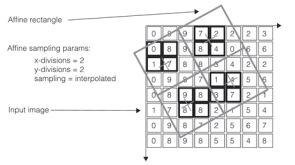
- Uninterpolated sampling is faster than both types of interpolated sampling.
- High-precision interpolation produces a more accurate image than bilinear interpolation; bilinear interpolation produces a more accurate image than uninterpolated sampling.

Depending on the platform you are using, the speed penalty for interpolated sampling can be very small.

Large Affine Pixel Sizes

No matter how large the affine pixels are in the affine sampling parameters, only those pixels in the input image that are adjacent to the centers of the affine pixels are considered when the sampled image is produced.

The figure below shows an example of this. Even though the affine rectangle covers a large area within the input image and bilinear interpolation is specified in the affine sampling parameters, only the values of the pixels with bold borders are considered in producing the sampled image.



Large affine pixels

In general, you should specify the size of the affine pixels to be relatively close to the size of the pixels in the input image.

Using the Affine Rectangle Sampling Tool

This section provides an overview of how you use the Affine Rectangle Sampling tool. For more information, you should refer to the documentation that is supplied with the programming interface you are using.

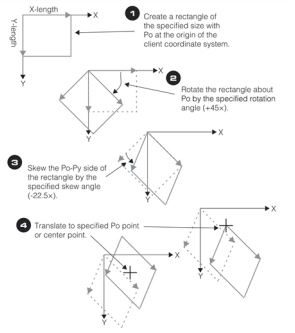
Steps to Using the Tool

In general, you will perform the following steps to obtain a sampled image:

1. Define an affine rectangle that captures the region of interest within the input image.
2. Specify affine sampling parameters that will produce an image with the desired number of pixels.
3. Apply the affine rectangle and sampling parameters to the input image.

Specifying an Affine Rectangle

You specify an affine rectangle by supplying all of the parameters required to define the rectangle to a function call or object constructor. The figure below shows how the constructor or function call uses the parameters you specify to construct an affine rectangle.

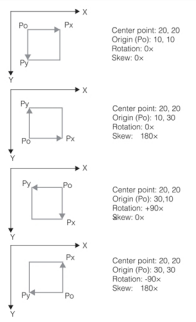


Creating an affine rectangle.

Handedness and Orientation

The steps shown in *Creating an affine rectangle*. on page 116 describe how to position an affine rectangle within a coordinate system. Controlling the orientation and handedness of the affine rectangle is done by specifying the rotation and skew values.

The figure below shows how different skew, rotation, and Po values determine the orientation and handedness of an affine rectangle.

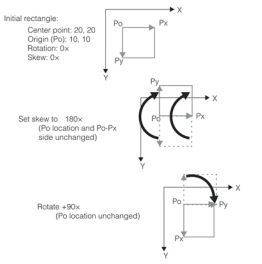


Creating an affine rectangle.

Note that each of the rectangles shown in the figure above occupies the same position within the coordinate system.

Changing Handedness

The figure below shows the specific steps required to change the handedness of an affine rectangle without moving the rectangle or changing the location of its origin (Po point).



Changing handedness requires changing skew and rotation

Note that if you use the CVL functions **ccAffineRectangle::xRotation()** and **ccAffineRectangle::skew()** to rotate and skew an existing affine rectangle, the rotation and skew are performed about the center point, not the origin point, as shown in the figure above. The result of the rotation and skew operations, however, would be exactly as shown in the figure above. (Also, note that you can apply the rotation and skew in either order.)

Choosing Orientation and Handedness

You specify the location of an affine rectangle to determine what portion of the input image is included within the rectangle. As shown in *Creating an affine rectangle.* on page 117, affine rectangles that occupy the same part of an image can have their vertex points in different locations.

The orientation and handedness of an affine rectangle determine how the affine rectangle is used.

- If you are specifying an affine rectangle to create a subsampled image, the image coordinate system of the subsampled image will reflect the orientation and handedness of the affine rectangle. *Computing a interpolated sample image* on page 112 shows how the affine rectangle vertices determine the image coordinate system of the sampled image.
- If you are specifying an affine rectangle as input to the Caliper or Projection tool, then those tools will use the Po-Py axis of the affine rectangle as the projection direction and the Po-Px axis as the search direction, as shown in *Unrotated projection region* on page 119.

Modifying an Existing Affine Rectangle

The CVL programming interface provides functions that let you modify certain characteristics of an existing affine rectangle. The table below summarizes these functions and their effects.

Function	Effect
ccAffineRectangle::center()	Translates rectangle to new center point. Orientation and size unchanged.

Function	Effect
ccAffineRectangle::cornerPo()	Translates rectangle to new origin point. Orientation and size unchanged.
ccAffineRectangle::xLength() **ccAffineRectanlge::yLength()**	Resizes rectangle without changing center point. Moves origin point, vertex locations. Orientation unchanged.
ccAffineRectangle::xRotation()	Rotates rectangle about center point to specified angle (client x-axis to Po-Px side). All vertex points will change. Width, height, and skew unchanged.
ccAffineRectangle::skew()	Skews rectangle. Center point fixed; all vertex points will change. Width, height, and rotation unchanged.

CVL functions to modify an affine rectangle

Projection Tool

The Projection tool transforms a two-dimensional image to a one-dimensional vector.

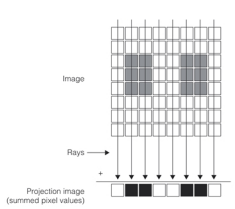
Projection Tool Overview

This section provides an overview of the Projection tool.

Projection

Projection is the process of reducing a two-dimensional image to a one-dimensional image. By constructing a one-dimensional projection of a two-dimensional image, you can greatly reduce the amount of time and storage required to process the image while retaining, or even enhancing, the information of interest in the two-dimensional image.

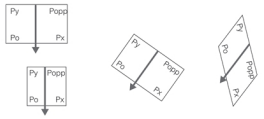
Projection works by summing the pixel values along parallel rays that lie in a particular direction within an image. The sum of the pixel values along each ray forms the value of a single pixel in the one-dimensional projection image. The figure below illustrates how a projection is formed.



Forming a projection

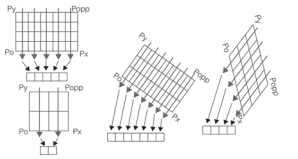
Defining Projection Regions

The region of an image that is summed to form a projection image is called the *projection region*. You specify projection regions as affine rectangles and affine rectangle sampling parameters. The projection direction within a projection region is always defined to be parallel to the Po-Py side of the affine rectangle. The figure below shows the projection direction defined for each of the affine rectangles shown in *Labeled affine rectangles* on page 111.



Projection direction of affine rectangles

The number of affine pixels along the x-axis of an affine rectangle determines the number of rays used to construct the projection image and the number of pixels in the projection image. The figure below shows the projection images that would be produced using each of the affine rectangles shown in *Affine rectangles and affine pixels* on page 111.



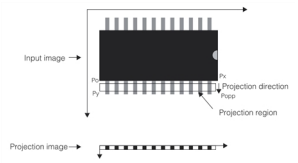
Projection images produced from affine rectangles

Note that the projection direction is always Po-Py and the *search direction* is always Po-Px.

Projection Region Examples

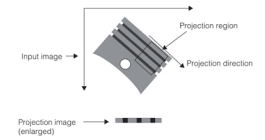
This section contains examples of different images and how you would specify a projection region to extract information about edges of interest.

The figure below illustrates an object where the thickness and spacing of leads on an integrated circuit package must be measured. Since the edges of interest are the edges of the pins, the projection direction is defined to be parallel to the pin edges. In order to maximize the strength of the edge information, the projection region does not extend beyond the ends of the pins. Because the object is not rotated with respect to the image, the projection region does not need to be rotated.



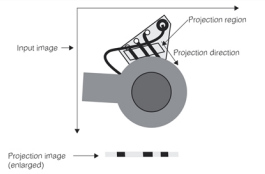
Unrotated projection region

The figure below illustrates an image where the required measurement is of the size and spacing of grooves in a part. Because the edges of interest are the sides of the grooves in the part, and because the part is rotated relative to the image coordinate system, the projection region must be rotated so that the projection direction is parallel to the grooves.



Rotated projection region

The figure below illustrates an object where the required measurement is of the thickness and spacing of three leads mounted on a part. Because of other features within the image, the projection region must be rotated in order for the projection direction to be parallel to the edges of interest and skewed to exclude portions of the image that do not contain the edges of interest.



Skewed and rotated projection region

How the Projection Tool Works

The Projection simply constructs a projection image by summing the values of column of affine pixels in the sampled image produced by the Affine Rectangle Sampling tool.

Specifying the Projection Region

You specify a projection region by specifying an affine rectangle and affine rectangle sampling parameters. When you project an image using the Projection tool, the tool constructs one ray for each column of affine pixels. The projection direction is parallel to the Po-Py side of the affine rectangle.

Projection direction of affine rectangles on page 118 and *Projection images produced from affine rectangles* on page 119 show how the affine rectangle and affine rectangle sampling parameters define the projection region and rays.

Clipping

The Projection tool offers the same options for handling clipping that the Affine Rectangle Sampling tool does. You can specify that the tool throw an error if clipping occurs, or you can specify that it return a weights image if clipping occurs.

The weights image returned by the Projection tool is a one-dimensional image (like the projection image). The value of each pixel in the weights image is equal to the total number of pixels that were summed to produce the corresponding pixel in the projection image.

If clipping occurs, not all pixels in the projection image are computed using the same number of pixels from the input image.

Projection Images

The projection images produced by the Projection tool have two special characteristics.

- They are images of 32-bit pixels
- They have a height of 1 pixel

The transformation object associated with a projection image lets you map points within the projection image to points within the input image. This lets you map edge peaks in a projection image to features in the input image.

Using the Projection Tool

This section provides an overview of how you use the Projection tool. For more information, you should refer to the documentation that is supplied with the programming interface you are using.

Steps to Using the Tool

In general, you will perform the following steps to use the Projection tool:

1. Define an affine rectangle that captures the edges of interest within the input image. Make sure the edges lie parallel to the projection direction.
2. Specify affine sampling parameters that will produce an image with the desired number of pixels.
3. Supply the affine rectangle, sampling parameters, and input image to the Projection tool.

Framework Differences

This section summarizes the differences between the OMI and CVL versions of the Image Transformation tools.

Tool Organization

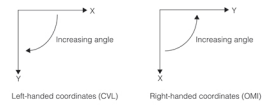
The CVL and OMI versions of the Image Transformation tools are organized differently. The table below shows how you gain access to the tools using OMI and CVL.

Tool	CVL API	OMI API	OMI GUI
Affine Rectangle Sampling Tool	*ch_cvl/affrect.h* *ch_cvl/affsampl.h* *ch_cvl/afftrans.h*	**AcuProces::Warp()**	**Proces** object's **Warping** tab
Projection Tool	*ch_cvl/project.h*	**AcuCaliper::GetProjResult()**	**Caliper** object's **Projection** tab

OMI and CVL image transformation tool interfaces

Coordinate Systems

Keep in mind that the CVL image coordinate system and the default CVL client coordinate system are both left-handed coordinate system while the OMI Image coordinate system and default World coordinate system are both right-handed coordinate systems. The figure below shows the difference between the coordinate systems.



OMI and CVL default coordinate systems

Polar Coordinate Transformation Tool

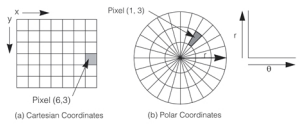
The Polar Coordinate Transformation tool transforms Cartesian coordinates to polar coordinates.

Polar Coordinate Transformation Overview

Cartesian coordinates are a natural choice for the digital representation of images: digital cameras are made of a rectangular grid of light-sensors, which maps well to the two-dimensional array in software. The pixels are square, and position is specified by horizontal distance (x) and vertical distance (y).

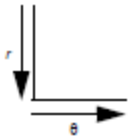
Polar coordinates, on the other hand, are based on a circular grid. The pixels look like wedges or keystones, and position is specified by radius (r) and angle-from-horizontal (θ). Thus, pixels are not uniform: as the distance from the origin increases, the pixels increase in area and change in shape.

The figure below compares the Cartesian and polar coordinate systems.

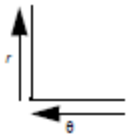


Comparison between Cartesian and polar coordinates.

You specify the resolution in both the radial and circumferential directions. In the above example, the radius represents the y-coordinate and is measured vertically in the positive direction; however, it can also be measured in the negative direction.



In the example, theta represents the x-coordinate and is measured in the positive direction; however, it can also be measured in the negative direction.



See also the class reference pages for **ccEllipseAnnulusSection**, *Selecting the Polar Image Area* on page 124.

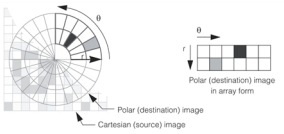
If the resolution is too coarse (too few pixels, compared with the source image), the polar coordinate tool will only sample a subset of the source pixels. If the resolution is too fine (too many pixels), the tool will interpolate new pixel values. In both cases there is the danger of creating false information. See *Selecting the Polar Image Area* on page 124 for more information.

How the Polar Coordinate Tool Works

The Polar Coordinate Transformation tool provides a method of transforming from a Cartesian to a polar representation. Mathematically speaking, these images should be identical. But because images are comprised of discrete pixels, errors are introduced in the interpolation or averaging required in the transformation.

The Polar Coordinate Transformation tool can be used alone for processing images with strong circular symmetry. Combined with the Cognex searching tools, this tool can be used as part of a more complex vision operation such as reading or verification.

Each pixel in the destination image can contain at most four source image pixels, as shown in the figure below. The destination image is the polar projection, or *rotation signature*, of the source. The Polar Coordinate Transformation tool calculates each polar pixel value by averaging the weighted values of the for corresponding pixels in Cartesian system. Because the pixels in the two images differ in shape, the transformation may result in some distortion.



Cartesian source image mapped to polar destination image.

Both the source image and the destination image are represented as **ccPelBuffers**. By convention, the polar radius is the y dimension and the polar angle is the x dimension of the **ccPelBuffer**.

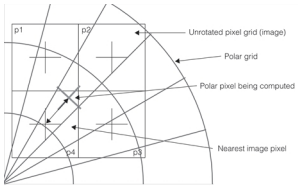
Sampling the Cartesian Source Image

The problem of determining the pixel values when sampling a Cartesian source image that is not aligned to the pixel grid of the input image is solved by *sampling* the pixel grid to produce the sampled image. The Polar Coordinate Transformation tool supports two types of sampling: *uninterpolated sampling* and *bilinear interpolation*.

Uninterpolated Sampling

Uninterpolated sampling computes the value for each pixel in the sampled image by determining the pixel value of the pixel whose center is closest to the center of the pixel. This is called *nearest neighbor* sampling.

The figure below shows how uninterpolated sampling works.

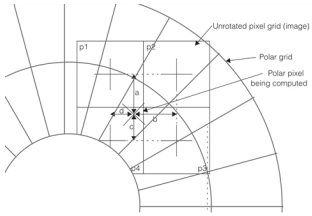


Uninterpolated sampling

Bilinear Interpolation

Bilinear interpolation considers the values of the *four* pixels closest to the center of each affine pixel in the affine sampling rectangle. The distance-weighted average of the values of the four pixels is computed and used for the value of the affine pixel.

The figure below shows how bilinear interpolation is computed.



Bilinear interpolation

The value of the interpolated polar pixel is computed using the following formula:

$$(1-a)\,(p2\times(1-b)+p1\times b)+a\,(p3\times(1-b)+p4\times b)$$

where

a, b, c, and d are the four distances shown in *Bilinear interpolation* on page 112, normalized so that the distance between pixel centers is 1.

p1, p2, p3, and p4 are the values of the four pixels shown in *Bilinear interpolation* on page 112.

The computed interpolated pixel values are rounded to the nearest whole integer value.

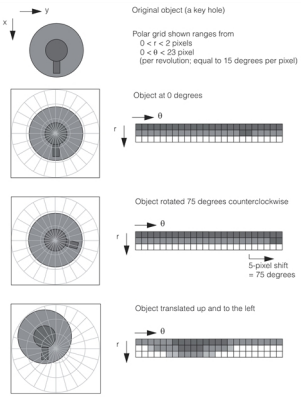
Note: Depending on which platform you are using, the interpolation arithmetic might be done using scaled integer math or floating point math. In general, this has no effect on the resulting image.

Effects of Rotation and Translation on Polar Transformation

The major property of a polar image is that it is insensitive to object rotation about its center, but intolerant of object translation in x-y space. The figure below shows an object and its rotation signature at 0 degrees rotation, at 75 degrees rotation, and with translation in x-y.

The rotation of an object about its center produces a polar image with the same shape but shifted in theta.

The translation of an object produces a polar image which does not resemble the original model. The polar routines are designed to be somewhat tolerant of poor center placement, but at the expense of execution speed. For this reason, the Polar Coordinate Transformation tool is often preceded by a center-finding function, such as blob analysis.



Object, at 0 degrees, rotated 75 degrees, and translated in x-y

Using the Polar Coordinate Transformation Tool

This section describes how to use the Polar Coordinate Transformation tool.

Cartesian Searching of a Polar Image

The Polar Coordinate Transformation tool is primarily used for Cartesian searching of a polar image, such as character reading or verification.

Some applications, such as the reading of characters printed in an arc, require the visual appearance of the object to be maintained. For applications such as these, the polar routines must produce a polar destination image that is as close as possible a representation of source image. In addition to 1:1 pixel mapping, the source may be spatially averaged to the destination by a given factor. For this application, the polar routines automatically calculate the size of the destination image.

Consider the example of text printed on a compact disc, shown in the figure below. The letters are curved along an arc of constant radius, so the search area is limited and the radius is known. One possible scenario is as follows:

- Find center of the compact disk.
- Find the position of the letters using low resolution polar searching. Define a polar destination image with 360 degree range between start angle and end angle.
- Transform the curved characters in Cartesian space into a straight line using a high resolution polar transformation tool. The source image area is determined by the previous step.
- Use character recognition to read or verify letters.

Example of search and character recognition of polar image

Selecting the Polar Image Area

The transformation to polar coordinates takes time and space to complete. You can minimize the amount of resources required by reducing the area of interest. The range should be as small as possible, but large enough to contain features of interest.

A polar sampling region is specified using the following parameters:

- An ellipse annulus section (**ccEllipseAnnulusSection**). This object describes an angular wedge, or section, of the annulus formed between two aligned ellipses with the same major/minor axis ratio.

- The number of X and Y samples. These parameters specify the size of the destination image and (equivalently) the number of divisions in the polar sampling grid. yNumSamples nominally specifies the height of the destination image and the number of samples in the radial grid direction. xNumSamples nominally specifies the width of the destination and the number of samples in the angular grid direction.
- An interpolation method selects how the image pixels are processed to generate the sample value at each sampling point. If you specify no interpolation, the sample value is the value of the single pixel whose center lies nearest to the sampling point. If you specify bilinear interpolation, the sample value is the bilinearly weighted average of the four pixels whose centers are nearest to the sampling point. See *Sampling the Cartesian Source Image* on page 122 for a description of interpolation methods.

You specify these parameters using a **ccPolarSamplingParams** object in CVL.

The function **cfPolarTransformImage()** maps pixels in the (Cartesian) source image to pixels in the (polar) destination image. Each pixel in the destination image contains some number of whole and/or fractional pieces of the source image pixels. The source pixels are averaged together and normalized to form the corresponding output pixel for the destination image.

Image Warp Tool

This chapter describes the Image Warp tool, a tool that transforms a distorted image into a corrected image. The transformation is done with a nonlinear transform known as the warp transform.

This section and *Some Useful Definitions on page 126* give an overview of the chapter and define some terms you will encounter as you read.

The chapter has the following major sections:

Image Warp Tool Overview on page 126 introduces the Image Warp tool and describes its purpose.

How the Image Warp Tool Works on page 127 describes how the Image Warp tool corrects distorted images.

Using the Image Warp Tool on page 130 describes the Image Warp tool C++ API.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

accuracy: A parameter you can set that will control the accuracy of the Image Warp tool mapping. Higher accuracy requires a larger mapping table in memory, and longer execution times.

calibration: A procedure to produce a transform that establishes the relationship between pixels in an image and physical units such as microns. Calibration transforms can also correct for distortion in an image.

destination mask: Part of the Image Warp tool that indicates which pixels in the destination window contain mapped pixels from the source image.

transform: A C++ class that can perform mapping between two coordinate systems. Most often used in CVL to map between image space and client (physical) space. Transform mapping can be linear or nonlinear.

warp: A term used by the Image Warp tool for the mapping a distorted source image to a corrected destination image.

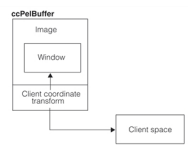
warp transform: The transform used to train the Image Warp tool.

window: A rectangular region of a root image in a pel buffer. A subset of the entire image.

Image Warp Tool Overview

All images contain some distortion caused by tolerances in the acquisition system electronics, and imperfections in camera lenses. In many applications the distortion does not affect the operation of vision tools, but for applications requiring high accuracy or applications with high distortion, it must be corrected to achieve the desired results.

The most common way to correct distortion is to build the correction into the client coordinate calibration transform. Using this method, vision tools work on the distorted image, but when results are transformed to real world units, the distortion correction is applied. See the figure below.



Client coordinate transform

For these applications the client coordinate transform can be linear (**cc2XformLinear**) or nonlinear (**cc2XformPoly**). These calibration transforms are discussed in *Calibration Tools on page 158* and the *Images and Coordinates* chapter of the *CVL User's Guide*.

Although this method works well in many cases, the vision tool algorithms are applied to a distorted image. A better solution is to create a tool that builds a distortion correction map, and then transforms distorted images into corrected images for vision tools to use. This is the purpose of the Image Warp tool. See the figure below.

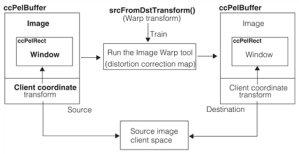
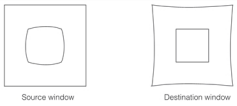


Image Warp tool overview

To use the Image Warp tool it must be trained with a warp transform and other information about the source pel buffer and the destination pel buffer. The warp transform can be created from calibration plate images and tools provided with CVL, or you can provide your own warp transform. During training the tool builds a pixel map from the source image to the destination image and also creates the destination client coordinate transform that transforms warped destination image coordinates to the correct client coordinates.

Once the tool is trained you run it on distorted images to correct the images, and then these corrected images can be used with other CVL tools such as Blob and Caliper to produce more accurate results. The figure below shows an example of a distorted image of a square, and its corrected image.



Warp correction example

How the Image Warp Tool Works

To use the Image Warp tool you must train it with a warp transform and other information about the source and destination pel buffers. You can use your own warp transform (*eGeneral* mode), or you can create a warp transform using CVL tools (*eUsingCalibration* mode). The warp transform you create with CVL tools can be created from intrinsic and extrinsic parameters, or from images of a calibration plate. The figure below shows an overview of the training options.

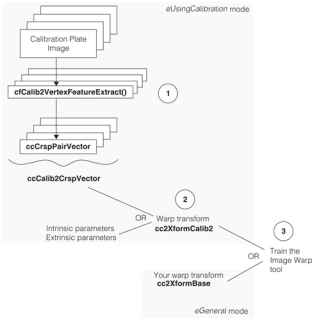


Image Warp tool training

The tool has a flexible API and can be trained in one of three ways. You can train it with a warp transform that you create yourself; this is called *eGeneral* mode and allows you to correct images using a variety of correction transforms. The *eUsingCalibration* mode accepts a calibration transform object created from the CVL class **cc2XformCalib2**. This transform can be created either from your system parameters or from calibration plate images. When you create a calibration object from images you must first extract feature points from the images and couple them with corresponding physical locations. When you use CVL tools, the global function **cfCalib2VertexFeatureExtract()** does this corresponding for you.

Two support tools were developed along with the Image Warp tool and the three parts are labeled 1, 2, and 3 in the figure above. The following is a brief description of each part;

1. The new global function **cfCalib2VertexFeatureExtract()**.

   This function analyzes an image of a checkerboard calibration plate and finds features (rectangle corners) and builds a correspondence table of feature locations in the image and feature locations in physical space.

2. The new calibration transform object, **cc2XformCalib2**.

   This new nonlinear calibration object was specifically designed to accept vectors of feature correspondences from the **cfCalib2VertexFeatureExtract()** global function, and to be used as a warp transform for the Image Warp tool. Note that since the **cfCalib2VertexFeatureExtract()** feature corresponder is separate from the calibration object, the option is open for other corresponders to be written. For example, a corresponder could be written for images of a grid-of-dots calibration plate and the corresponder output used to create **cc2XformCalib2** objects.

   Note also that **cc2XformCalib2** objects can be very accurate nonlinear transforms you may wish to use with other CVL tools besides the Image Warp tool. Creating **cc2XformCalib2** calibration objects is discussed in *Feature Correspondence Calibration* on page 164.

3. The Image Warp tool.

   The Image Warp tool was designed to be trained with a **cc2XformCalib2** warp transform. However, in *eGeneral* mode you can train the tool with any linear or nonlinear transform derived from **cc2XformBase**. This leaves the door open for designers with special requirements to train the tool using a wide variety of warp transforms.

Once the tool is trained, you can run it repeatedly by calling **warp()**.

Training the Image Warp Tool

You train the Image Warp tool to prepare the tool for removing distortion from input images. You need train the tool only once for it to process many images in the same way. Training ahead saves execution time in your production environment. Before training the tool you provide it with the following information:

- A warp transform the tool uses to create the pixel map that maps the source image to the destination image. You can provide a CVL-generated warp transform (type **cc2XformCalib2**) or you can provide your own warp transform. Users who have their own proprietary calibration procedures may wish to create their own warp transforms, either linear (type **cc2XformLinear**), nonlinear (type **cc2XformPoly**), or some composition of these two.

  When you provide a warp transform created with CVL tools you also provide a point in the source image where the tool takes the scale from the source image client coordinate transform. This scale is used throughout the destination client coordinate transform so that distorted image features will be approximately the same size as corrected image features.

  Along with the source image point, you specify an *undistort* mode for the warp transform. The tool default enables all distortion correction modes, however, you can selectively disable any or all of the modes if your application requires it. The following distortion modes can be selectively enabled or disabled:

  - Perspective distortion:
    This is the distortion that causes squares to appear as quadrilaterals.
  - Radial distortion:
    Radial distortion causes the image edges to either bow out, or to be drawn in towards the image center.
  - Rotational distortion:
    The rotation of the xy-axes together in a plane.

- The source window and the destination window.
  These windows are of type **ccPelRect** and are applied to the source and destination pel buffers at run time. The windows restrict the image pixels that are mapped from the source to the destination. Since the Image Warp tool mapping is compute and memory intensive, you can speed up your application and use less memory if you only map the image area you need to work with. If the source and destination windows are not the same, only the overlapping section of the windows is actually mapped. (See *Image Warp Tool Results* on page 129).
- The source to destination mapping accuracy.
  The transform you use to train the tool dictates how accurate the distortion correction will be. The mapping table created at training time, however, dictates the accuracy of the transform implementation. Building a larger mapping table produces better accuracy but uses more memory and takes longer to run. You can control these parameters by specifying the accuracy you need in your application. During training the tool maps pixels from the source to the destination and then maps the points back into the source. If pixel locations mapped back differ from the original pixel locations by more than the *accuracy* you specify, the tool enlarges the mapping table and tries again. When all pixels pass the accuracy test the tool saves the mapping table in its trained state.

Running the Image Warp Tool

The Image Warp tool must be trained before you can run it. When you run the tool you pass it a source pel buffer containing a distorted image and a destination pel buffer where the tool is to place the undistorted image. When the tool finishes, you can use the destination pel buffer with other CVL tools such as Blob and Caliper. These tools should obtain more accurate results using the undistorted image.

Image Warp Tool Results

The primary result from running the Image Warp tool is the destination pel buffer which is a standard CVL class of type **ccPelBuffer<c_UInt8>**. Use any members of this class to investigate the result as you wish.

The Image Warp tool contains only one result you may be interested in; the destination mask. During training the tool computes the trained source window, trained destination window, and the destination mask from the source window and destination window you specified. The trained windows represent the overlap of the source window and the destination window. See the example in the figure below.

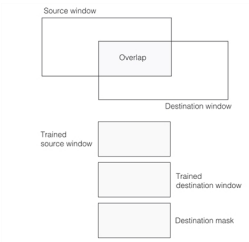


Image Warp tool destination mask

The trained source window specifies the source image pixels to be transformed. The trained destination window specifies where in the destination image the transformed pixels will be placed. Because the warp mapping is generally nonlinear, its common that some destination window pixels do not correspond to source window pixels. The destination mask shows this correspondence. Destination mask pixels values are either 255 or 0. A 255 indicates a source pixel is mapped into the destination at this location, and a 0 indicates one is not.

Destination mask example

In this example the warp transform corrects for barrel-type radial distortion which causes the destination image to be drawn inward towards the center. No source image pixels map to the drawn-in edges so 0-value pixels appear at the edges in the destination mask as shown.

Using the Image Warp Tool

The Image Warp tool is a class of type **ccImageWarp** that you program to remove distortion from a source image, producing an undistorted destination image. To use the tool you first train it as a preliminary step, and then you can run the tool repeatedly on as many images as you wish. The following sections provide information about how you program the tool in a C++ application.

Training the Image Warp Tool

You train the Image Warp tool by calling **ccImageWarp::train()** with no arguments. Prior to calling **train()** you must setup the tool object by calling the following setters:

- **srcFromDstTransform()**

  Call this setter to pass in the warp transform. You can use a CVL-generated warp transform (type **cc2XformCalib2**) or you can provide your own warp transform either linear (type **cc2XformLinear**), nonlinear (type **cc2XformPoly**), or some composition of **cc2XformLinear** and **cc2XformPoly**.

  CVL provides support for creating **cc2XformCalib2** transforms. This is discussed in *Feature Correspondence Calibration* on page 164.

  When you train with a **cc2XformCalib2** warp transform you also pass in a scale point (*pt*) and an undistort mode as follows:

  pt
  Specifies a point in the source image where the scale part of the source image client coordinate transform is taken. This scale will be used throughout the destination image client coordinate transform created by the Image Warp tool so that distorted image features will be approximately the same size as corrected image features.

  undistortMode
  Specifies which distortion corrections to make. The value you pass must be some combination of the flags defined in the **ccImageWarp::UndistortMode** enum. The default sets all flags and performs all corrections.

  The undistort modes are described in *Training the Image Warp Tool* on page 128.

- **srcRect()**

  You pass in a **ccPelRect** that defines the window in the source image that you wish to warp (remove distortion from). You can warp the entire source image if you wish, however, you will use less memory and your application will run faster if you limit the window only to the image area where your application applies CVL vision tools.

  This source image window is applied to the source image pel buffer each time you run the tool.

- **dstRect()**

  You pass in a **ccPelRect** that defines the window in the destination image where the tool places the corrected image. Keep in mind that the final destination window is the overlap of the window areas defined by **srcRect()** and **dstRect()**.

  This destination image window is applied to the destination image pel buffer each time you run the tool.

- **accuracy()**

  Call this setter to change the accuracy used for training. The default is 0.001 pixels. This setting will impact the accuracy of your distortion correction, and the performance of the Image Warp tool. See the description in the section *Training the Image Warp Tool* on page 128.

To train the Image Warp tool do the following:

1. Create a default Image Warp tool object, type **ccImageWarp**.
2. Set the warp transform you wish to use by calling **ccImageWarp::srcFromDstTransform()**.
3. Set the source window (**srcRect()**) and destination window (**dstRect()**).
4. Set the accuracy you wish to use (**accuracy()**).
5. Call **ccImageWarp::train()** to train the tool.

Running the Image Warp Tool

To run the Image Warp tool call **ccImageWarp::warp()** and pass in a source image pel buffer (type **ccPelBuffer_constr<c_UInt8>**) and a destination image pel buffer (type **ccPelBuffer<c_UInt8>**). On return the destination pel buffer contains the corrected image that you can use as input to other CVL vision tools.

Scene Angle Finder-II Tool

This chapter describes the Scene Angle Finder-II tool, software that determines the predominant angle of the features in an image.

Note:
The functionality implemented by this tool is similar to that implemented by the Scene Angle Finder tool described in the next chapter.

In general, the Scene Angle Finder-II tool provides better accuracy and performance than the Scene Angle Finder tool.

This chapter contains the following sections:

Scene Angle Finder-II Tool Overview on page 132 provides an overview of the Scene Angle Finder-II tool.

How the Scene Angle Finder-II Tool Works on page 134 provides a description of how the Scene Angle Finder-II tool works.

Using the Scene Angle Finder-II Tool on page 136 describes how to use the Scene Angle Finder-II tool in your application.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

parallel features: Linear features that do not intersect.

orthogonal features: Linear features that intersect at right angles.

unidirectional evaluation: The evaluation of features that appear in a single direction.

bidirectional evaluation: The evaluation of orthogonal features.

resolution pyramid: A series of images at increasing resolutions evaluated by the Scene Angle Finder-II tool. It is an inverse pyramid, with the lowest spatial resolution (smallest pixel count) at the top of the pyramid, and the highest spatial resolution (largest pixel count) at the bottom, or base of the pyramid.

pixel aspect ratio: The ratio of horizontal pixel size to vertical pixel size.

quality score: The ratio of the signal strength to the background noise, expressed in decibels (dB).

accept threshold: The minimum acceptable quality score.

Scene Angle Finder-II Tool Overview

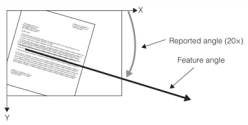
The Scene Angle Finder-II tool evaluates an image and returns the angle of the strongest features in that image.

The Scene Angle Finder-II tool measures angle by evaluating the image's linear visual features. It is intended for use on images with a predominance of parallel or orthogonal features. The Scene Angle Finder-II tool can evaluate features that appear in a single direction, such as the lines of text on a printed page, or can combine orthogonal features, such as the perpendicular lines composing the grid on a semiconductor wafer, into a single measurement.

The Scene Angle Finder-II tool will, on request, measure the angles of multiple independent features within a scene.

Unidirectional Evaluation

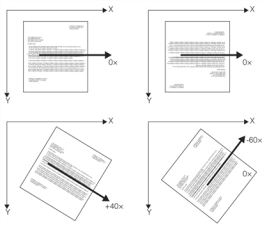
When run in unidirectional mode, the Scene Angle Finder-II tool evaluates the features in an image and returns the angle of the strongest features it finds. The figure below shows a low resolution image containing a portion of a page of text. The Scene Angle Finder-II tool reports the angle of the lines of text shown in the figure below as 20° because of the predominance of features in that direction.



Low resolution image of a page of text rotated 15°

Angle Range for Unidirectional Evaluation

When performing a unidirectional evaluation, the Scene Angle Finder-II tool reports angles in the range of -90° to +90°. The figure below shows examples of how the Scene Angle Finder-II tool reports angles in unidirectional mode.



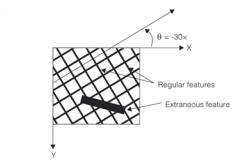
Angles returned for unidirectional evaluation.

Bidirectional Evaluation

When you specify *bidirectional evaluation*, the Scene Angle Finder-II tool measures the angle of the predominant orthogonal features in an image by evaluating the strength of features at θ, and the strength of the features at θ+90°. The product of the *quality scores* of the features (described below) is calculated to form a final quality score, so that orthogonal features reinforce each other and non-orthogonal features are suppressed, in the sense that they receive quality scores lower than those of orthogonal features.

You should not use bidirectional evaluation in an attempt to ensure that parallel lines without orthogonal components are not selected as representing the angle of the scene. While the quality scores of orthogonal features are enhanced, the quality scores of non-orthogonal features are not reduced to 0; in fact, the quality score of a bold non-orthogonal feature may surpass that of very faint orthogonal features.

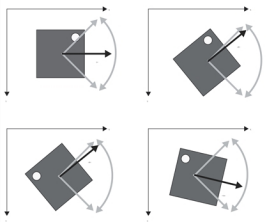
The figure below shows an image of a grid with strong features in two directions and a random feature represented by the thick dark line. The Scene Angle Finder-II tool returns a value of -30°, indicating strong features at -30° and +60° (90° out of phase with -30°). If only one result were requested, the random feature in the figure below would not be found as an orthogonal feature because of the lack of features out of phase with it. If multiple results were requested, it is likely that the random feature would be found as an angle in the scene.



Grid with strong features in two directions

Angle Range for Bidirectional Evaluation

When performing a bidirectional evaluation, the Scene Angle Finder-II tool reports angles in the range of -45° to +45°. *Angles returned for unidirectional evaluation.* on page 133 The figure below shows examples of how the Scene Angle Finder-II tool reports angles in unidirectional mode The light grey arrows and arc show the possible range of returned values.



Angle range used for bidirectional evaluation

How the Scene Angle Finder-II Tool Works

This section presents the operating parameters in detail, and discusses speed and accuracy issues related to the Scene Angle Finder-II tool.

Pyramid Resolution Evaluation

The Scene Angle Finder-II tool makes a preliminary evaluation of an image at reduced resolution, followed by a refined evaluation at increased resolution. This refinement may be repeated to the full resolution of the image. This technique greatly reduces the execution time with no degradation in the accuracy of the measurement.

The series of images at increasing resolution is called a *resolution pyramid*. It is an inverse pyramid, with the lowest resolution (smallest pixel count) at the top of the pyramid, and the highest resolution (largest pixel count) at the bottom, or base of the pyramid.

You can specify subsampling values for the initial and final resolutions. A subsampling value of 1 means that each pixel in the image is considered; no sampling is performed. Subsampling values greater than 1 indicate that the image is being subsampled. A subsampling value of 2 means that the image is reduced in size by a factor of 2 in both the x and y direction.

You should choose values based on the image contrast, number of edges in the image, and size of the features. The higher the contrast, the greater the number of edges, and the larger the features, the larger the subsampling values you should choose.

For optimum execution time, the initial subsampling value should be set as high as possible. However, setting this value *too* high may result in failure to find the feature(s) of interest. The final subsampling value should be set according to the desired accuracy. Setting this value too low when evaluating an image that has low contrast or few edges may cause instrumentation effects (described below) and noise to dominate, and a false value of 0° may be returned.

Accuracy and Speed

The Scene Angle Finder-II tool operates on any size image, evaluating the image's entire content. The accuracy of the measurement is proportional to the image size: the larger the image used, the more accurate the angle measurement (assuming the object of interest fills the image).

You can estimate the maximum error that the Scene Angle Finder-II tool experiences by first computing the *angle step*. The Scene Angle Finder-II tool works by evaluating the image features at a series of different angles. The angle step is the distance between angles being evaluated. If the angle step is 10°, then the Scene Angle Finder-II tool will evaluate the image at 36 different angles.

The angle step is determined by the size of the image being evaluated. The following formula gives the angle step for an image: $$anglestep = atan\left(\frac{2}{max\,(width,height)}\right)$$

where *width* is the width of the image in pixels and *height* is the height of the image in pixels. Note that width and height are the subsampled width and height of the image.

The maximum error experienced by the Scene Angle Finder-II tool is given by the following formula: $$maxerr \approx \frac{anglestep}{4}$$

The execution time of the Scene Angle Finder-II tool is proportional to the angle range specified and the size of the image.

You should keep the following points in mind as you specify subsampling values.

- If you specify too large a subsampling value for the initial resolution evaluation, the tool may fail to detect features that fall between the angles being evaluated.
- If you specify too large a subsampling value for the final resolution evaluation, the accuracy of the reported results is reduced.
- The ultimate accuracy of the tool's results is affected by image content. Images with curved or indistinct features will not generate as accurate results as images with distinct, linear features.

Accuracy in Bidirectional Mode

When performing a bidirectional evaluation of an image, the image should be square. Because the features are judged by the product of the quality scores (described below) in each direction, an image that is not square could unevenly weight one direction or the other, resulting in an inaccurate measurement.

Instrumentation Effects at Zero Degrees

Due to a variety of causes such as video noise, acquired image may contain artifacts and instrumentation effects that always give a weak signal at 0°. If the image is of high contrast and the feature being evaluated contains many edges, the instrumentation effects are not significant. However, if the image is of low contrast, or the feature has relatively few edges, the instrumentation effects can dominate. This causes the Scene Angle Finder-II tool to report 0° as the strongest feature. Half resolution or subsampled images reduce the instrumentation effects to the point that they are not a problem in most scenes.

Pixel Aspect Ratio

The aspect ratio is defined as the ratio of horizontal pixel size to vertical pixel size. If you use the Scene Angle Finder-II tool to analyze images that were acquired using a camera of frame grabber that produced nonsquare pixels, the accuracy of the returned angle can be reduced.

The Scene Angle Finder-II tool uses the input image's client coordinate system when it determines scene angle. If you use the Calibration tool to construct a transformation that correctly describes the transformation between a real-world calibration grid and an image's image coordinate system, then set the image's image-to-client transformation to be this transformation, this will correct for any pixel aspect ratio.

Scoring

During the preliminary phase of the image analysis, the Scene Angle Finder-II tool computes a raw score for all possible angles in the image. Angles with raw scores above a raw score threshold that you specify are evaluated during the final phase, and a final *quality score* is computed. Angle results with quality scores above the accept threshold that you specify are returned by the tool.

The *quality score* is the ratio of the signal strength to the background noise, expressed in decibels (dB). The minimum acceptable *quality score* is called the *accept threshold*. A typical value is 2.0 dB, as this value is high enough to differentiate a feature from its neighbor. Cognex recommends that you do not use values below 0.5 dB.

Note that the higher a value you specify for the accept threshold, the faster the tool will run. If the threshold is set too high, however, the tool might not return any results.

Multiple Results

The Scene Angle Finder-II tool will returns as many results as you request. You can use this capability to determine the angles of multiple features in an image.

Using the Scene Angle Finder-II Tool

The table below provides an overview of the parameters you supply to the Scene Angle Finder-II tool.

Parameter	Notes
Angle span	The range of scene angles of interest, specified as a start angle and an end angle The start angle must be less than the end angle, and the angle range can be no more than 45 ° for bidirectional mode and 90° for unidirectional mode.
Number of results	How many results to return. If you are searching for multiple features at different angles, you should specify a number of results equal to the number of features. Results are returned from strongest to weakest.
Initial and final subsampling values	The subsampling factor to apply to the image before the first (low-resolution) and final (high-resolution) pass. Higher subsampling values reduce execution but may miss features.
Accept thresholds	In addition to the accept threshold, which specifies the quality score value that a result must have to be returned, you can also specify a low-resolution accept threshold. This threshold is used to exclude raw results from further evaluation.
Mode	Specify unidirectional mode to search for linear features at any orientation. Specify bidirectional mode to search for orthogonal features (linear features at 90° angles from each other).

Scene Angle Finder-II tool parameter summary

Scene Angle Finder Tool

This chapter describes the Scene Angle Finder tool, software that determines the predominant angle of the features in an image.

Note:
The functionality implemented by this tool is similar to that implemented by the Scene Angle Finder-II tool described in the previous chapter.

In general, the Scene Angle Finder-II tool provides better accuracy and performance than the Scene Angle Finder tool.

This section gives an overview of the chapter and defines some terms you will encounter as you read.

Scene Angle Finder Tool Overview on page 137 provides an overview of the Scene Angle Finder tool.

How the Scene Angle Finder Tool Works on page 140 provides a description of how the Scene Angle Finder tool works.

Using the Scene Angle Finder Tool on page 141 describes how to use the Scene Angle Finder tool in your application.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

edge: Boundary between two regions of different pixel values

edge angle: Orientation of an edge at an edge pixel, expressed as the angle between the client coordinate system x-axis and a line drawn perpendicular to the edge going from dark to light

edge angle image: Image created by replacing each pixel in an input image with a pixel whose value is equal to the edge angle of the pixel in the input image

edge magnitude: Strength of an edge, expressed as the difference between pixel values on the two sides of the edge

edgelet: Data structure describing the sub-pixel location of an edge along with its angle and magnitude

fold value: The number of times a histogram is folded upon itself

reinforcing edges: Two or more edges at equal angular intervals

Scene Angle Finder Tool Overview

The Scene Angle Finder tool measures the angle of the predominant features within the scene. You typically use the Scene Angle Finder tool to determine the angle of a scene so that you can correct the angle using the Affine Rectangle Sampling tool, or so you can supply appropriate parameters for PatMax, Caliper or other vision tools.

Scene Angle Finder and Scene Angle Finder II

The Scene Angle Finder tool provides an alternate implementation of the capabilities of the Scene Angle Finder-II tool. The key differences between the Scene Angle Finder-II and Scene Angle Finder tools are listed below:

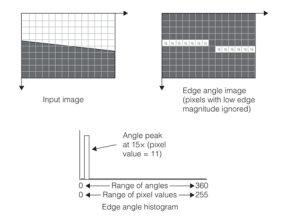
- Instead of offering a choice between *unidirectional* and *bidirectional* operating modes, the Scene Angle Finder lets you specify a *fold value*. A configurable fold value lets you control exactly how the tool reports reinforcing edges.

- Instead of reporting angles in the range of -90° through +90° reports angles in the range 0° through 360°.
- The Scene Angle Finder tool, while generally less accurate than the Scene Angle Finder II tool, may be faster.

How Scene Angle is Determined

The Scene Angle Finder tool uses the Edge tool to compute an edge angle image of the input image that you supply. The Scene Angle Finder computes a histogram of the edge angle image. Peaks in this histogram correspond to the angle of the scene.

The figure below shows a simple example of how the Scene Angle Finder tool would determine the angle of a scene. In this case, the scene is a simple horizon.

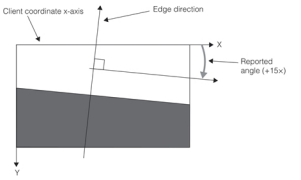


Computing scene angle

Angles in edge angle images (and edge angle histograms) are always scaled to the range 0 to 255. The angles reported by the Scene Angle Finder tool are always expressed in degrees.

The angles reported by the Scene Angle Finder tool are defined to be the angle between the client coordinate system x-axis and an angle which is normal to the edge and in the positive edge gradient direction (from dark to light) plus 90°. This angle corresponds to the angle of the predominant feature boundary

In the example shown in the figure above, the edge angle is computed as shown in the figure below.



Reported scene angle is edge angle + 90°

Scenes Without Predominant Angles

The angle of scenes which do not include a predominance of straight edges are difficult to determine using the Scene Angle Finder tool. The figure below shows the edge angle histogram produced from a scene without a predominance of straight lines. Because of the absence of straight lines, there are no peaks in the edge angle histogram, and consequently no way to determine the angle of the scene.

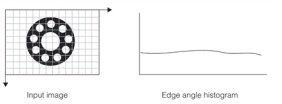
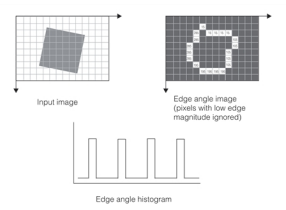


Image with no predominant angle: no edge angle histogram peaks

Reinforcing Edges

Most scenes which are appropriate for the Scene Angle Finder tool have multiple, reinforcing edges. For example, if a scene consists of one or more square features, the Scene Angle Finder tool will detect four predominant scene angles,

as shown in the figure below.



Reinforcing edges

The Scene Angle Finder tool detects four peaks, one for each of the four edges of the square feature. By default, the Scene Angle Finder tool returns four angles. Because the four peaks occur at a regular angular interval, they are called reinforcing edges.

You can control how the Scene Angle Finder handles scenes with *reinforcing edges* in two ways.

- You can specify a *fold value*.
- You can specify an *angle span*.

Each of these techniques is discussed in the following sections.

Fold Value

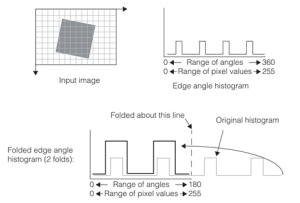
You can combine the peaks generated by reinforcing edges by specifying a fold value. The fold value controls how many times the overall edge angle histogram is folded upon itself.

If your scene includes square features, then there are four reinforcing edge angles. To treat all four of these angles as a single scene angle, you would specify a fold value of 4.

When you specify a fold value, the Scene Angle Finder tool folds the edge angle histogram the number of times you specify. This has two effects:

- Reinforcing edge angle peaks are overlaid on top of each other (summed).
- The number of histogram bins required to describe a particular angle is increased, improving the resolution of the Scene Angle Finder tool.

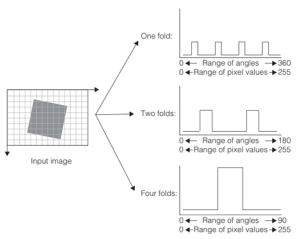
The figure below shows how a fold value of 2 affects the edge angle histogram generated in *Reinforcing edges* on page 139.



Folding the edge angle histogram

Note that the peaks in the folded edge angle histogram in *Folding the edge angle histogram* on page 139 are both higher and wider than the peaks in the unfolded edge angle histogram. The peaks are higher because more edge pixels contribute to each peak and wider because more histogram bins are being used to depict a given range of angles.

You can specify any number of folds. The figure below shows the effect of specifying 1, 2, and 4 folds on the edge angle histogram generated in *Reinforcing edges* on page 139.

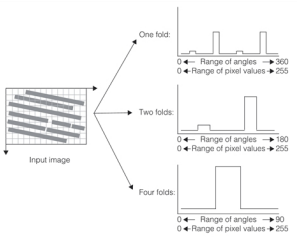


Effect of different fold values

While both a fold value of 2 and a fold value of 4 produce distinct peaks, in the case of a square feature, 4 is usually the correct value. This is because the two peaks in the 2-fold histogram are indistinguishable.

Note that in the figure above the peak location in the 0° to 90° range is no more meaningful than the two peaks in the 0° to 180° range or the four peaks in the 0° to 360° range.

For scenes with a predominance of horizontal or vertical features, a fold value of 2 may produce a more meaningful result, as shown in the figure below.



Fold value for rectangular features

The features in the figure above are not reinforcing in all four directions. Accordingly, the scene angle is meaningful in the 0° to 180° range, not just in the 0° to 90° range, and the 2-fold histogram is more meaningful than the 4-fold edge angle histogram.

Angle Span

You can also limit the range of angles which the Scene Angle Finder tool will consider. This range of angles is called the *angle span*.

Specifying an angle span causes the Scene Angle Finder tool to ignore peaks in the edge angle histogram that lie outside the angle span you specify.

If you specify an angle span and a fold value, the angle span must lie within the range of angles produced by the fold value you specify.

How the Scene Angle Finder Tool Works

This section describes how the Scene Angle Finder tool works. The Scene Angle Finder follows these basic steps to compute the scene angle:

- It smooths and sub-samples the image.
- It computes an edge angle histogram of the image.
- It filters the edge angle histogram.
- It computes and returns the locations of peaks in the edge angle histogram.

Each of these steps is described in the following sections.

Smoothing and Sub-Sampling

The Scene Angle Finder tool smooths and sub-samples the input image.

Smoothing accentuates and strengthens regular, rectilinear features, and it tends to diminish the effect of small randomly oriented features.

Sub-sampling reduces the number of input pixels that must be processed and can reduce execution time.

You can control smoothing and sub-sampling parameters used by the Scene Angle Finder tool.

Computing the Edge Angle Histogram

After smoothing and subsampling the image, the Scene Angle Finder tool determines the angle of the features in the image.

Contrast Threshold

You can specify a contrast threshold. Only pixels in the edge angle image which correspond to edge magnitudes greater than the threshold you specify are considered by the Scene Angle Finder tool.

Computing the Edge Angle Image

The Scene Angle Finder tool uses the Edge tool to compute the edge angle image. For information on the edge tool, see the chapter *Edge Tool* on page 207.

Supplying an Edgelet List

You can also supply your own edgelet list, as generated by the Edge tool. Edgelets are subpixel edge locations.

If you supply an edgelet list, the Scene Angle Finder tool performs no smoothing or subsampling, and it does not apply a contrast threshold.

Histogram Filtering

For most real images, the edge angle image histogram will contain many small peaks in addition to the larger peaks of interest. You can specify that the Scene Angle Finder tool apply a simple 1-dimensional filter to the histogram.

Angle Span

If you know the approximate or expected scene angle or range of scene angles, you can specify an angle span. The Scene Angle Finder tool only computes and returns results for peaks within the range of angles you specify. The default angle span is -90° to +90°.

Scene Angle Finder Tool Results

For each edge peak in the filtered edge angle histogram, the Scene Angle Finder tool returns the angle and peak magnitude of the result.

Using the Scene Angle Finder Tool

The table below provides an overview of the parameters you supply to the Scene Angle Finder tool.

Parameter	Notes
Angle span	The range of scene angles of interest, specified as a start angle and an end angle If either the start angle or end angle are greater than 180°, then angles are returned in the range 0° to 360°. Otherwise, angles are returned in the range -180° to 180°

Parameter	Notes
Fold value	The number of times to fold the histogram upon itself The fold value corresponds to the number of reinforcing edge angles within a 360° range.
Filter width	The filter size to apply to the edge angle histogram, specified in histogram bin units
Contrast threshold	The minimum edge magnitude required before an edge is added to the edge angle histogram
Subsampling and smoothing values	The parameters used to smooth and sub-sample the image

Scene Angle Finder tool parameter summary

Frame Averaging

This chapter describes the Frame Averaging tool, which allows you to accumulate a number of captured camera images of the same scene, and return an average image. Averaging several images of the same scene may have the effect of attenuating visual noise from the camera's sensor, potentially resulting in cleaner images. The cleaned-up image can then be used as input to other CVL vision tools.

This chapter has the following sections:

Some Useful Definitions on page 143 defines some terms you will encounter as you read.

Averaging Images on page 143 describes why and how to average images, and describes the standard and rolling mode methods of image averaging.

Some Useful Definitions

averaged image: A computed pel buffer, derived by taking each pixel position in a frame average buffer and dividing the summed pixel value in each pixel position by the number of images accumulated.

frame average buffer: An object of type **ccFrameAverageBuffer** into which you accumulate a number of pel buffers. Each pixel in the frame average buffer is the sum of the numerical pixel values of the corresponding pixel position in the pel buffers added to the frame average buffer. The values accumulated in the frame average buffer are either computed from each pel buffer added (standard mode), or from the last N pel buffers added (rolling average mode).

Averaging Images

The Frame Averaging tool is an API that allows you to accumulate multiple images of the same camera scene into a frame average buffer, and to compute the average image or a standard deviation image. This tool is of value when averaging different image captures of the same non-moving scene; the images are expected to be as identical as possible.

The averaged image is computed as the sum of the numerical pixel values for each pixel position in a set of images, divided by the number of images.

The standard deviation image is computed, for each pixel position in a set of images, as the root mean square difference between the sum of pixels in the set and its corresponding average value.

When you average a number of same-scene images, you can effectively attenuate the effects of noise from the camera's sensor. This is especially useful in environments where the light source is variable or flickering. Averaged images can be cleaner images than raw camera images, and can be presented to other CVL vision tools to produce better results.

There is no guarantee than an averaged image is a cleaner image. The average taken is a simple mathematical average of the pixel values at each position. However, in the usual case, for image captures of an identical scene with fairly stable lighting, an averaged image should show less video noise.

Frame Averaging Modes

The Frame Averaging tool has two modes:

- Standard
- Rolling Average

In standard mode, you accumulate a number of pel buffers into a frame average buffer, then ask the tool to calculate an average image on demand. In this mode, you have the option of accumulating statistics that let the tool calculate a standard deviation image from your accumulated pel buffers. If enabled, you can request the standard deviation image in addition to, or instead of, the averaged image.

In rolling average mode, you specify a number (N) of images to accumulate in the frame average buffer. The frame average buffer then maintains an average of the last N images accumulated. This provides an efficient way to maintain a rolling average of the last N images in a continuos image stream. In this mode, the average image cannot be retrieved until at least N images have been added to the frame average buffer. The standard deviation calculation is not available in rolling average mode.

Creating a Frame Average Buffer

Create a frame average buffer as an instance of the **ccFrameAverageBuffer** class:

```
ccFrameAverageBuffer <imgPel, accPel> &buffer;
```

The template parameters *imgPel* and *accPel* specify the pel type of the images being accumulated and of the accumulator, respectively, using values like the instantiations provided for **ccPelTraits**: **c_UInt8**, **c_UInt16**, and so on. The *accPel* size must always be larger than the *imgPel* size, to leave room for pixel value summing.

Note: The **ccFrameAverageBuffer** class is currently implemented only for the case where *imgPel* = **c_UInt8** and *accPel* = **c_UInt16**.

To create the frame average buffer *frameAvg*, use code like this example:

```
ccFrameAverageBuffer <c_UInt8, c_UInt16> frameAvg;
```

Using Standard Mode

In standard mode, you accumulate a fixed number of pel buffers into a frame average buffer. When those images are accumulated, you request a averaged image, or standard deviation image, or both.

Standard mode might be appropriate, for example, on an assembly line where the part to be inspected is moved into place and held motionless for a known time period. Instead of taking one camera image while the part is in place, you can take several images and average them.

The standard deviation image may help you troubleshoot lighting problems, or other image quality problems, by showing you parts of the image that are more at variance than others. The limiting case is an all black standard deviation image, which means that no difference was found between the set of images being averaged. Any lighter portions of your standard deviation image are indicators of difference between the accumulated images.

The following example shows the use of image averaging in standard mode.

```
#include <ch_cvl/vp8100.h>
#include <ch_cvl/acq.h>
#include <ch_cvl/prop.h>
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/constrea.h>
#include <ch_cvl/windisp.h>
#include <ch_cvl/frameavg.h>

int cfSampleMain(int, TCHAR** const)
{
 cc8100m& fg = cc8100m::get(0);
 const ccStdVideoFormat& fmt =
 ccStdVideoFormat::getFormat(_T("Sony XC75 640x480"));
 ccStdGreyAcqFifoPtrh fifo = fmt.newAcqFifo(fg);

 fifo->triggerModel(cfManualTrigger());
 fifo->triggerEnable(true);
```

```
// Create a frame average buffer and set it to accumulate
// standard deviation statistics.
ccFrameAverageBuffer<c_UInt8,c_UInt16> frameAvg;
frameAvg.setStandardMode(true);

// Set up window titles and sizes
const TCHAR* title1 = cmT("Single Image Capture");
const TCHAR* title2 = cmT("Average of 10 Images");
const TCHAR* title3 = cmT("Standard Deviation of 10 Images");
ccIPair vga = ccIPair(320+20, 240+20);
ccIPair start = ccIPair(10,70);
ccIPair offset2 = ccIPair(367,0);
ccIPair offset3 = ccIPair(0,378);

// Obtain and display a single image for comparison.
fifo->start();
ccAcqFailure fail;
ccPelBuffer<c_UInt8> pb = fifo->complete(&fail);
if (fail) {;} // Insert code to report any acquisition failure
ccDisplayConsole display1(title1, start, vga);
display1.image(pb);
cfWaitForContinue(10,5);

// Now take 10 images and add each to the frame average buffer
for (int i = 1; i <= 10; ++i)
{
// Start the next acquisition.
fifo->start();

// Use complete() to obtain a pel buffer from the last start().
ccAcqFailure fail;
ccPelBuffer<c_UInt8> pelbuf = fifo->complete(&fail);

if (fail) {;} // Insert code to report any acquisition failure

// Add this image to the frame average buffer.
frameAvg.add(pelbuf);
}

// Obtain and display the averaged image.
ccDisplayConsole display2(title2, (start + offset2), vga);
ccPelBuffer<c_UInt8> avg = frameAvg.average();
display2.image(avg);
cfWaitForContinue(10,5);

// Obtain and display the standard deviation image.
// If there was no variation between the images added to
// the frame average buffer, then the standard deviation
// image is all black.
ccDisplayConsole display2(title2, (start + offset3), vga);
ccPelBuffer_const<c_UInt8> stddev = frameAvg.stdDevImage();
display3.image(stddev);
```

```
 cfWaitForContinue(10,5);

 return 0;
}
```

Notice that **average()** returns a **ccPelBuffer<c_UInt8>**, while **stdDevImage()** returns a **ccPelBuffer_const<c_UInt8>**.

Using Rolling Average Mode

In rolling average mode, you set a number (N) of images to accumulate in the frame average buffer. Each image added is automatically averaged with the previous images. When N images have accumulated, but not before, you can request the averaged image.

The maximum number of images to average, N, must be a power of 2 between 2 and 256, inclusive.

You might use rolling average mode in cases where you are pulling images in a continuous stream out of an acquisition FIFO. Newly added images continuously accumulate in the frame average buffer. Once N is reached, if the buffer is not reset, **frameAvg. numFrames()** stays at the count of N as new images are added. Thus, you can request an averaged image anytime after the first N images. This obtains an average of the last N images in the FIFO. Your application is responsible for ensuring that the last N images are acquisitions of the same scene under the camera.

Another way to use rolling average mode is to reset the frame average buffer after every N images, request the averaged image, and then let the count of newly added pel buffers build up again to N. In this case, to determine when the rolling average buffer has reached N images, you can compare:

```
frameAvg.numFrames() // the number of frames accumulated
```

to

```
frameAvg.maxNumRollingFrames() // the number N to accumulate
```

Once you have requested the averaged image, reset the frame average buffer so that **numFrames()** can start accumulating again:

frameAvg.reset();

The following example shows how you might use rolling average mode, demonstrating the case of resetting the frame average buffer after each N images.

```
#include <ch_cvl/vp8100.h>
#include <ch_cvl/acq.h>
#include <ch_cvl/prop.h>
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/constrea.h>
#include <ch_cvl/windisp.h>
#include <ch_cvl/frameavg.h>

int cfSampleMain(int, TCHAR** const)
{
 cc8100m& fg = cc8100m::get(0);
 const ccStdVideoFormat& fmt =
 ccStdVideoFormat::getFormat(_T("Sony XC75 640x480"));
 ccStdGreyAcqFifoPtrh fifo = fmt.newAcqFifo(fg);

 fifo->triggerModel(cfManualTrigger());
 fifo->triggerEnable(true);

 // Create a frame average buffer and set it to generate an
 // average image when eight images have been added to the buffer.
```

```
 ccFrameAverageBuffer<c_UInt8,c_UInt16> frameAvg;
 frameAvg.setRollingAverageMode(8);

 // Set up window titles and sizes
 const TCHAR* title1 = cmT("Single Image Capture");
 const TCHAR* title2 = cmT("Average of 8 Images");
 ccIPair vga = ccIPair(320+20, 240+20);
 ccIPair start = ccIPair(10,70);
 ccIPair offset = ccIPair(367,0);

 // Obtain and display a single image for comparison.
 fifo->start();
 ccAcqFailure fail;
 ccPelBuffer<c_UInt8> pb = fifo->complete(&fail);
 if (fail) {;} // Insert code to report any acquisition failure.
 ccDisplayConsole display1(title1, start, vga);
 display1.image(pb);
 cfWaitForContinue(10,5);

 // Now take 24 images and add each to the frame average buffer.
 // The 24 is for illustration only; in production, this might be
 // a while(0) or some other continuous mechanism.
 for (int i = 1; i <= 24; ++i)
 {
 // Start the next acquisition.
 fifo->start();

 // Use complete() to obtain a pel buffer from the last start().
 ccAcqFailure fail;
 ccPelBuffer<c_UInt8> pelbuf = fifo->complete(&fail);

 // Insert code here to report any acquisition failure.
 if (fail) {;}

 // Add this image to the frame average buffer.
 frameAvg.add(pelbuf);

 if ( frameAvg.numFrames() == frameAvg.maxNumRollingFrames() )
 {
 // Obtain and display the averaged image.
 ccDisplayConsole display2(title2, (start + offset), vga);
 ccPelBuffer<c_UInt8> avg = frameAvg.average();
 display2.image(avg);
 cfWaitForContinue(10,5);
 // Reset the frame average buffer so it can accumulate
 // another rolling average image.
 frameAvg.reset();
 }
 }
 return 0;
}
```

Color Tools

This chapter describes the Color Segmenter tool, the Color Match tool, and the Composite Color Match tool. It contains the following sections:

Some Useful Definitions on page 148 defines some terms that you will encounter as you read.

Color Tools Overview on page 148 provides an overview of the color tools.

Color Segmenter Tool on page 150 describes the Color Segmenter tool.

Color Match Tools on page 153 describes the two Color Match tools.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

color space: A 3-dimensional coordinate space used to define and locate individual color values. Common spaces include RGB (red-green-blue) and HSI (hue-saturation-intensity).

color wheel: A circle used to represent all possible values for hue within the HSI color space.

segmentation: Division of the pixels in an image into object pixels and background pixels. Segmentation is an essential part of blob analysis.

Color Tools Overview

The CVL color vision tools described in this chapter provide several capabilities:

- They segment color input images into object and background pixels based on color. You then use the segmented image as input to a grey-scale vision tool.
- They match and classify color samples. You can use this capability directly to identify or classify objects in images based on color.

These capabilities are provided by three color tools:

- The *Color Segmenter tool* uses regions of color space to map color images into grey scale images that can be processed using grey-scale vision tools.
- The *Color Match* tool lets you define a set of reference colors, then use the tool to determine which of the reference colors a run-time input color is closest to. You can specify the run-time input color values directly, or you can provide an image and have the tool compute the average color, then match it. The Color Match tool matches colors based on minimizing the distance in color space between two color values.
- The *Composite Color Match* tool does the same thing as the Color Match tool, except that it matches colors based on the distribution of pixel values rather than simple color definitions. You train the tools's reference colors by providing an image for each color. To run the tool, you supply an image and the tool determines which of the reference color images most closely matches the run-time color image, based on the distribution of pixel values in the two images.

Each of these tools is described separately in this chapter.

Note: You can use the RSI Search tool (see *RSI Search on page 324*) to locate patterns in color images.

Understanding Color Spaces

Most systems for encoding colors use three-dimensional coordinate systems, where each axis describes a separate aspect of the color. When the values from each of the axes are combined, a particular color is produced. These

coordinate systems are also called *color spaces.*

Two commonly used color spaces are RGB (Red-Green-Blue) and HSI (Hue-Saturation-Intensity). Each space is described in this section.

RGB Color Space

Red, green, and blue are called the *primary colors* because by combining light of these three colors in different proportions, light of any color can be produced. The RGB color space encodes colors by specifying the amount of red, green, and blue light required to produce a given color.

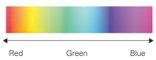
The RGB color space closely corresponds to how color images are acquired by color video cameras and displayed by color monitors. Color video cameras separate the light from the scene into the red, green, and blue components and transmit each of these signals separately. Color display monitors accept separate red, green, and blue signals, then mix red, green, and blue light to reproduce the colors in the original scene.

Because of its close relationship to how color is recorded and displayed by video technology, the RGB color space is a very common way of encoding colors. One limitation of the RGB color space is that it is often difficult to relate a particular RGB value to how a color appears. For example, the color with a value of R-105, G-206, and B-102 is a pale green.

HSI Color Space

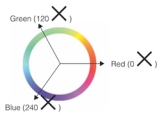
When human beings describe colors, they often do so in terms of the color's *hue* (is it red, yellow, or violet), *saturation* (is it a pure color or is it diluted with white), and *intensity* (is it bright or dim). The HSI color space is based on the fact that all perceivable colors can be described in terms of these three measures.

- The hue of a color corresponds to its location within the visible portion of the electromagnetic spectrum of radiation:



Hue

The conversion of a hue to a numerical value is done by creating a *color wheel*. A color wheel places all of the hues from the spectrum on a circle. A particular hue is specified by giving the angle at which that hue lies on the color wheel. The figure below shows a color wheel.



Color wheel

Note: The range of hue values is scaled to the range 0 through 255.

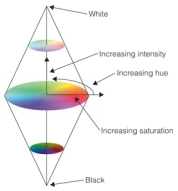
- The saturation of a color is the degree to which the color is mixed with grey or white. A fully saturated color is pure. As a color becomes mixed with white (or grey) it becomes less saturated. For example, as the color red becomes less saturated (more mixed with white), it tends toward pink. The red of a fire engine is a saturated red, while the pink of a carnation is an unsaturated red. The figure below shows a range of saturation values for a single hue.

Saturation

A color's saturation is expressed as a number from 0 through 255 that indicates the amount of dilution by white. A value of 0 for saturation indicates grey.

- The intensity of a color is a measure of its brightness. The intensity of a color is defined as the average value of the red, green, and blue components that make up the color.

  The three components that make up the HSI color encoding can be visualized together as making up a space composed of two cones, as shown in the figure below.



HSI Color Space

  All possible colors occupy points within this volume. Completely saturated colors exist on the outside surface of the space. Shades of grey are located along the central axis of the shape; black is at the bottom vertex and white is at the top vertex.

Comparing RGB and HSI Color Spaces

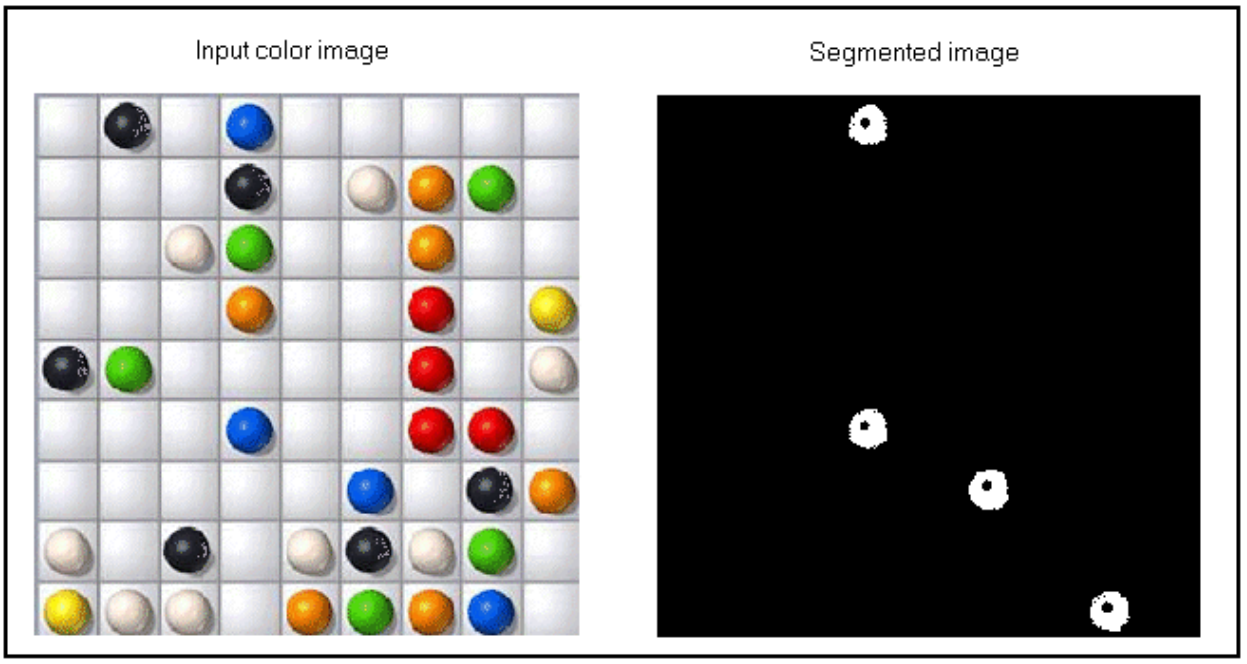
Both the RGB and HSI color spaces can describe any possible color; each color space has advantages and disadvantages:

- Because color images are acquired and displayed using RGB technology, manipulating RGB images can be faster and more efficient than using other color spaces.
- The HSI color space is modeled on how human beings perceive and describe colors. It can be easier to associate colors with their HSI values than with their RGB values.
- The transformation between RGB and HSI is nonlinear; converting an RGB to an HSI image takes time.

Color Segmenter Tool

The Color Segmenter tool analyzes a color image in order to produce a grey scale image consisting of light pixels against a dark background, where the light pixels correspond to features from the color input that fell into one or more color ranges that you specify. The grey scale image a Color Segmenter tool produces represents only those features of the color image you are interested in, and can be further analyzed with another vision tool, such as a Blob tool.

For example, the figure below shows a pattern of colored spheres and shows how a Color Segmenter tool can be used to isolate the blue ones:



Color segmentation

This output image can be further analyzed with a Blob tool to generate such information as the number of blue spheres, their location, their size, and so on.

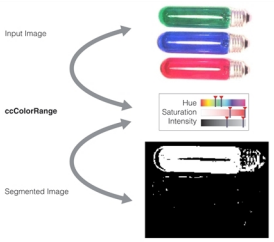
Color Space

A Color Segmenter tool can accept images in both RGB and HSI color space. Since the goal of a Color Segmenter tool is to define one or more ranges of desirable color, you may find it easier to work with images defined in the HSI color space, since this allows you to narrowly define the specific hue you want to target. Once the proper Hue has been identified, adjusting the allowable range for Saturation and Intensity can often allow the Color Segmenter tool to reliably locate the features you want to analyze in the output grey scale image.

Color Ranges

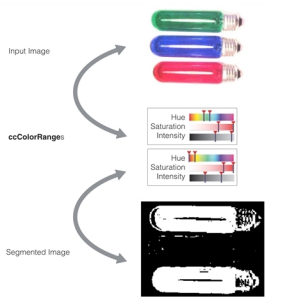
A Color Segmenter tool allows you to take a color image and separate those features that exhibit a desirable range of color. In order to perform this segmentation, you must define one or more ranges of color from a reference image, or several reference images if necessary, of the objects you want to analyze with your vision application. As the tool operates it selects any pixel within an enabled range for the segmented output image.

For example, the figure below shows an input image and the segmented image it generates with a single color range for locating the green light:.



Segmentation with one color range

The same Color Segmenter tool can be configured with a second color range for locating both the green and red lights, as shown in the figure below.



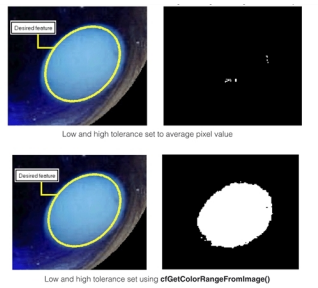
Segmentation with two color ranges

When you specify multiple color ranges, the tool applies each range in turn to the input image, then computes a single segmentation image where each pixel has the maximum value of any of the intermediate segmentation images (a logical OR of the images).

Tolerance Values

The color range you specify includes a low tolerance, a high tolerance, and a fuzzy tolerance for pixel values in each axis of the color space. You can specify these values yourself, you can use the average pixel value for the feature of interest, or you can use the **cfGetColorRangeFromImage()** functions defined in *ch_cvl/coltool.h* to compute them automatically, based on a histogram analysis of the image or image region that you supply.

In most cases, as shown in *Segmentation with one color range* on page 151, the histogram-based tolerances provide good results.

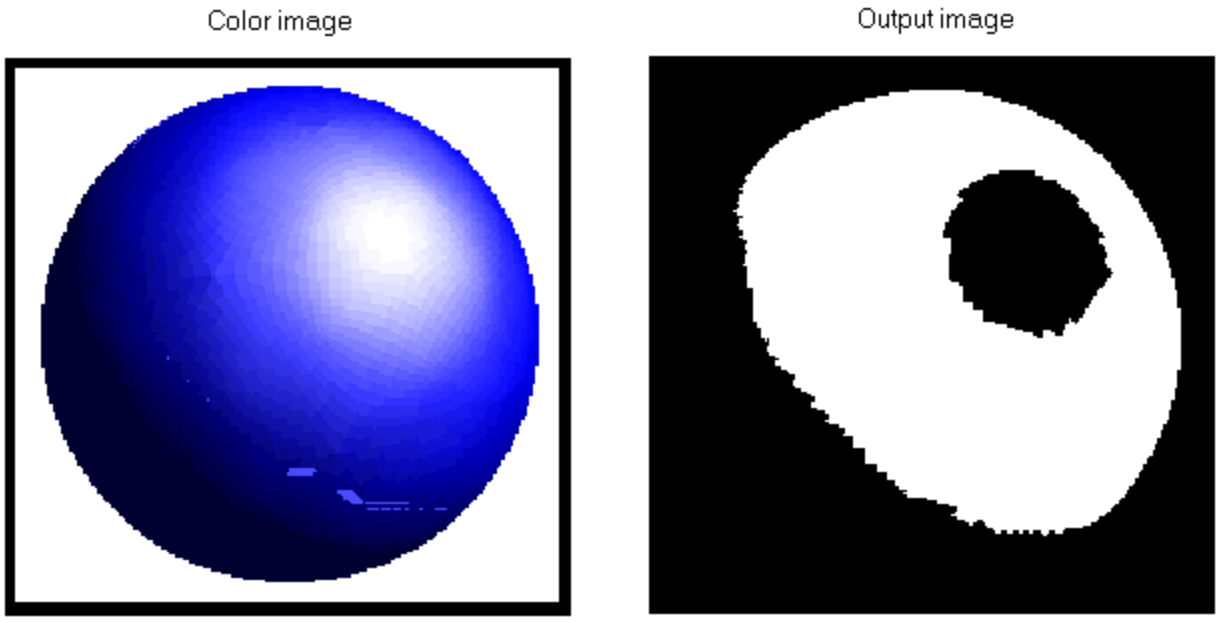


Segmentation tolerance values

Softness

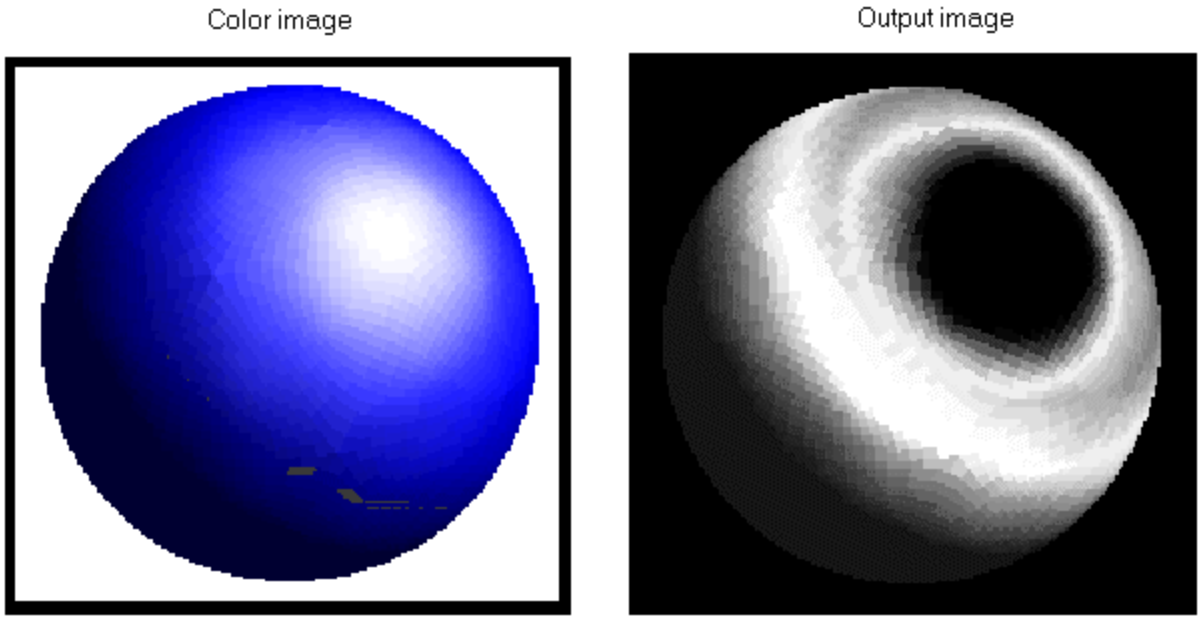
By default, a Color Segmenter tool generates a grey scale output image where pixels having a value of 255 represent features from the original color image you are interested in examining further while pixels with a grey value of 0 represent all other features.

In some applications, it is not only necessary to distinguish the color of a feature of interest, but also to represent the degree of the color in the output image. For example, the figure below shows a color image and the output image it can generate when configured to use hard threshold values:



Hard threshold

If an application needed more information on the value of various shades of blue in the color image, the settings of the Color Segmenter tool can use soft thresholding to generate an output image with various degrees of grey values, where blue features that are close to the defined range generate lighter grey values and features that are farther from the defined range generate darker grey values. For example, the figure below shows the same color image and the output image it can generate when the tool is configured to use a soft threshold



Soft threshold

You specify soft thresholding by supplying a single fuzzy threshold value; the tool creates a symmetric soft threshold at both the low and high threshold values.

Color Match Tools

The color match tools compare a region of color in a run-time image against a set of reference colors, and they generate a set of scores to indicate how closely the area of the run-time image matches each known color. The higher the comparison score, the greater the similarity. The tools also return the color from the reference set that is the best match with the color observed in the run-time image.

The Color Match tool uses a simple distance algorithm to match colors based on the distance in color space between the sample color and the colors in the reference set. The Composite Color Match tool matches colors based on the similarity in the distribution of colors between the sample image color and the reference colors. Unlike the Color Match tool, the Composite Color Match tool requires a separate training step where you train each of the reference colors.

Both tools are described in this section.

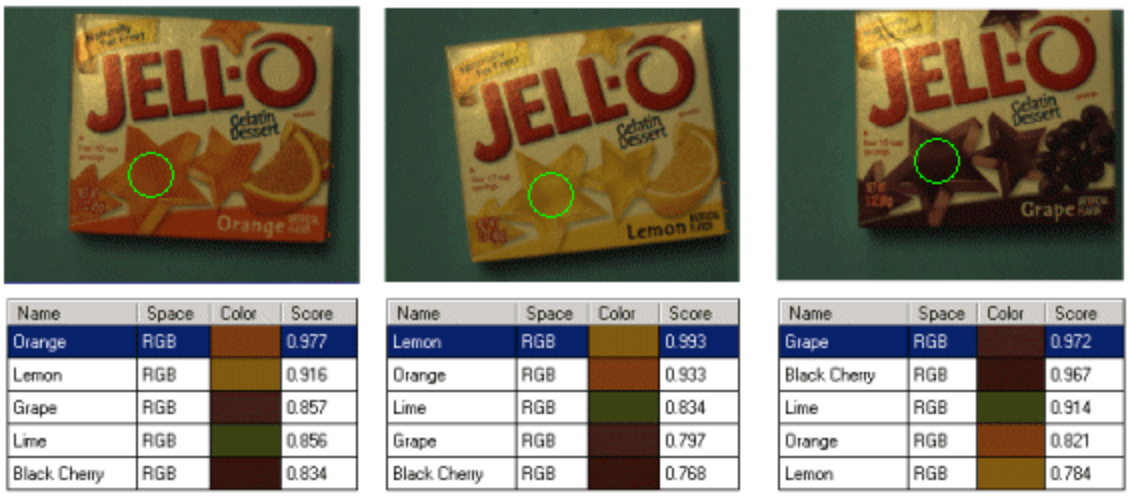
Color Match Tool

The Color Match tool compares a region of color in a run-time image against a set of reference colors, and generates a set of scores to indicate how closely the area of the run-time image matches each known color. The higher the comparison score, the greater the similarity. The tool returns the index of the color from the reference set that represents the highest match with the color observed in the run-time image.

For example, The figure below shows a series of images that need to be distinguished by color:

Color images requiring classification

The figure below shows how the Color Match tool could be used to classify these images using five reference colors.



Name	Space	Color	Score
Orange	RGB		0.977
Lemon	RGB		0.916
Grape	RGB		0.857
Lime	RGB		0.856
Black Cherry	RGB		0.834

Name	Space	Color	Score
Lemon	RGB		0.993
Orange	RGB		0.933
Lime	RGB		0.834
Grape	RGB		0.797
Black Cherry	RGB		0.768

Name	Space	Color	Score
Grape	RGB		0.972
Black Cherry	RGB		0.967
Lime	RGB		0.914
Orange	RGB		0.821
Lemon	RGB		0.784

Classified images

Reference Colors

A Color Match tool compares a region of a run-time image against a set of reference colors and determines which reference color generates the best match. You specify the reference colors as **ccColorValue** objects that simply define the color's location within the selected color space.

To run the tool, you invoke the **cfColorMatch()** global function, passing it a vector of reference colors and either a single **ccColorValue** to match or an image and region to match. If you supply an input image and region, the tool computes the average color for the region and matches that.

Matching Colors

A Color Match tool generates a match score based on the color distance between the color of the run-time image and the reference color. The tool uses the following formula to generate the color distance:

$$\textbf{Color distance} = \text{sqrt}\left(\sum w_i * (runtime\_color_i - reference\_color_i)^2\right)$$

Where w_i is the weight for the i_{th} component of the color value.

It computes a match score based on this color distance; the lower the distance, the higher the score. The score is in the range 0.0 through 1.0.

The formula is applied identically to both RGB and HSI images. The ***i*** component represents values for Red, Green, and Blue in the RGB color space or the values for Hue, Saturation, and Intensity in the HSI color space.

If necessary, you can alter the weight given to any single component before the tool calculates the color distance. This can be useful in situations where two reference colors are very similar and the tool does not always distinguish between them correctly in run-time images. The figure below shows how increasing the weight of the intensity component can help distinguish between two colors with similar hue and saturation values:

Adjusting weights to improve discrimination

Note: When computing the color distance for the Hue color plane in HSI space, the Color Match tool takes into account the fact that Hue values wrap from 255 back to 0.

Results

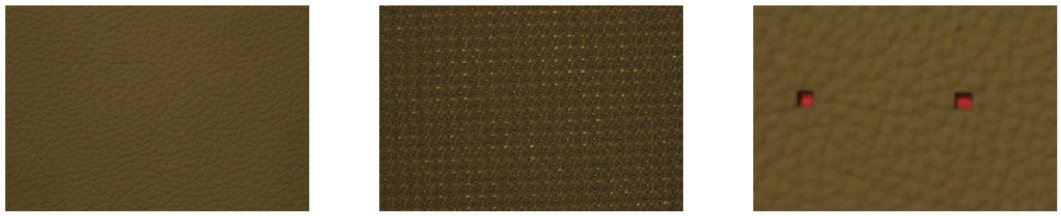
The Color Match tool generates the following results:

- The index of the reference color with the highest score
- The score and a vector of scores for all reference colors
- A confidence score, between 0 and 1, calculated as: (highest score - second highest score) / (highest score + second highest score). The confidence score indicates how well the color with the highest match score can be distinguished between other reference colors. Low confidence scores indicate a narrow range of reference colors.
- The number of reference colors used for matching

Composite Color Match Tool

The Composite Color Match tool examines the color content of an area in an image against a set of trained reference colors, and generates a set of scores to indicate how closely the area of the run-time image matches reference color.

The Composite Color Match tool is intended for applications where the colors being matched are spatially non-uniform. For example, the figure below shows a series of textured patterns that could be matched using the Composite Color Match tool.



Textured color images

Unlike the Color Match tool, which simply computes the distances between pairs of ccColorValue objects in a color space, the Composite Color Match tool examines the distribution of color pixels in the run-time image and then computes a similarity score between the run-time image and each trained reference color.

Note: Unlike the Color Match tool and the Color Segmenter tool, the Composite Color Match tool only works with images defined in the RGB color space.

Training Reference Colors

Unlike the Color Match tool, the Composite Color Match tool requires that you train reference colors from image data. You train the tool by supplying a vector of:

- Images,
- Images and regions, or
- Images and masks

In addition to the reference images to train, you can also supply two training parameters:

- A Gaussian Smoothing kernel size
- A subsampling percentage

Each of these parameters is described in this section.

Gaussian Smoothing

Gaussian smoothing removes some detail and image noise from a composite color. You can enter a value between 0 and 24 to indicate the square size of the filter the smoothing operation will use. Smoothing each image sample can reduce image noise introduced by various sources and can make the comparison more reliable in vision applications where image noise is a factor. Be aware, however, that there is a trade-off between the amount of smoothing you can perform and the accuracy of any comparison.

Sampling Percentage

A sampling percentage will allow the Composite Color Match tool to use a sample of each original image you use to generate the comparison score, which can improve the execution speed of the tool at the cost of some accuracy.

Running the Tool

The Composite Color Match tool has two run-time parameters.

- You can specify that the tool normalize the intensity of run-time images before matching. This can improve the reliability of the tool when used in situations where scene brightness changes between images.
- You can specify a matching accuracy value (a value greater than 0.0 and less than or equal to 1.0). Specifying a matching accuracy less than 1.0 increases the tolerance value used by the tool to determine whether two colors match. Specifying lower matching accuracy values can increase the tool's speed but reduces the accuracy of the match.

Results

The Composite Color Match tool generates the same results as a Color Match tool.

Calibration Tools

This chapter describes the calibration tools that help you create calibration transforms that relate image space coordinates to physical space coordinates.

This section and *Some Useful Definitions on page 158* give an overview of the chapter and define some terms you will encounter as you read.

This chapter includes the following major sections:

Calibration Tools Overview on page 159 provides an overview of the calibration tools provided with CVL.

How the Calibration Tools Work on page 162 provides a description of how each calibration tool and how it works.

Using the Calibration Tools on page 168 tells you how to use the calibration tools.

Checkerboard Calibration Plates on page 171 includes specifications for a checkerboard calibration plate for use with the Feature Correspondence Calibration tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

2D client coordinate system: A 2D coordinate system that has its x-y axes in the image plane. It is described by pixels coordinates and the image client transform. If the image client transform is an identity, this coordinate system is same as the image coordinate system.

2D physical coordinate system: A rectangular 2D coordinate system with its x-y axes in the plane of a planar physical object observed through the camera.

Grid2D and Plate2D physical coordinate systems: Rectangular 2D coordinate systems with their x-y axes in the plane of a planar physical object observed through the camera. The planar physical object is typically a checkerboard calibration plate. The origin of Grid2D is the same as that of Plate2D, and the axes of Grid2D have the same orientation as the axes of Plate2D:

- The origin of the coordinate systems is at one of the vertices of the checkerboard calibration plate.
- The directions of the x-axis and y-axis are along the lines through a row of vertices and column of vertices, respectively.

The difference is that the unit of measurement is in tiles for Grid2D and in physical units (defined by the pitch of the calibration plate checkers) for Plate2D.

Grid2D is used primarily to express the position of a Data Matrix code in the checkerboard grid (in tiles).

Plate2D is used primarily to express the physical position of a vertex point in the run-time image (for example, in millimeters). For each vertex point, correspondence is established between its Client2D position and Plate2D position.

3D physical coordinate system: A rectangular 3D coordinate system that is an extension of 2D physical coordinate system described above. A positive z-axis in the direction pointing away from the camera is added.

3D camera coordinate system: A rectangular 3D coordinate system with its positive z-axis along the optical axis of the camera, pointing towards the observed object.

calibration: A procedure to create a transform that establishes the relationship between points in one coordinate space and corresponding points in another coordinate space.

calibration plate: A precision manufactured flat plate used to calibrate image acquisition systems.

checker: A square on a checkerboard calibration plate.

checkerboard vertex: A point on a checkerboard or a checkerboard image that is a shared corner of two white and two black checkers.

client coordinates: Image coordinates transformed by the client coordinate transform that is part of the pel buffer. For many applications this client coordinate transform is an identity transform, in which case image coordinates are the same as client coordinates.

Data Matrix: A 2D matrix symbology made up of modules arranged in a square or rectangle within a perimeter finder pattern.

extrinsic parameters: Properties of an image acquisition system that are external to the camera.

feature: An identifiable shape in an acquired image. For example, a dot in an image of a grid-of-dots calibration plate.

feature correspondence: Matching feature locations in an image to corresponding locations in physical space.

feature extractor: A software routine that scans a calibration plate image and returns a list of found feature locations and their corresponding physical coordinates.

grid pitch: The distance, in physical units, between adjacent checkerboard vertices.

intrinsic parameters: Properties of an image acquisition system that are internal to the camera and optics.

linear transform: A transform defined by a (2x2) matrix and a (2x1) vector.

nonlinear transform: A transform created from a set of data points. The transformation varies depending on the data point you transform. It is not consistent over a range of points as is the case of a linear transform.

physical coordinates: The physical space coordinate system. For example, the coordinate system used to locate points on a calibration plate.

pitch: The x and y distance between elements in a calibration plate. For example, the width ($Pitch_x$) and height ($Pitch_y$), in physical units, of the rectangles on a checkerboard calibration plate.

residual error: The difference between the actual grid point locations and the locations predicted by applying the calibrated transformation to the known grid spacing.

transform: A C++ class that can perform mapping between two coordinate systems. Most often used in CVL to map between image space and client (physical) space. Transform mapping can be linear or nonlinear.

transformation object: An object containing functions that map points in one coordinate system to another coordinate system. For example, a mapping between image space and client coordinate space. The transformation can be linear or nonlinear.

Calibration Tools Overview

A calibration tool creates calibration objects (transforms) that relate locations in image space to locations in physical space. You typically use these calibration objects as the client coordinate transform in pel buffers but they can be used for other purposes also, such as the transform that defines the warping for the Image Warp tool.

CVL provides two different and distinct calibration tools: the *grid-of-dots* calibration and the *feature correspondence* calibration. The grid-of-dots calibration is the older tool that uses an image of a grid-of-dots calibration plate to compute a linear transform (type **cc2XformLinear**) or a nonlinear polynomial transform (type **cc2XformPoly**). This tool has one class that does everything from the feature extraction to transform calculation and uses only grid-of-dots calibration plates.

The newer Feature Correspondence Calibration tool creates a nonlinear transform (type **cc2XformCalib2**) that models radial optical distortion and perspective distortion. The tool has two components: feature extraction and calibration. The feature extractor is specifically designed to extract features in an image of a checkerboard calibration plate The calibration component can create a calibration transform using the output of the feature extractor. This two step design allows you to create a **cc2XformCalib2** calibration object from your own set of features if you choose not to use the

Cognex checkerboard feature extractor. For example, you can use any set of corresponding image points and physical points to create a **cc2XformCalib2** calibration object using this calibration tool.

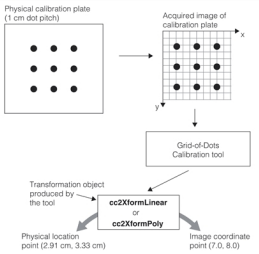
The **cc2XformCalib2** calibration object has been found to be more accurate than the grid-of-dots calibration objects for various distortions such as perspective distortion.

The *grid-of-dots* calibration and is discussed in the following section and the *feature correspondence* calibration and is covered in *Feature Correspondence Calibration* on page 160.

Grid-of-Dots Calibration

The Grid-of-Dots Calibration tool constructs a transformation object that relates locations in physical space, as defined by a grid-of-dots calibration plate, to locations in an image. You use the tool by acquiring an image in which the features are at known spacing. You supply the tool with the spacing of the features *in physical coordinates* and the tool constructs a transformation object that maps points between image coordinates and physical coordinates (also known as client coordinates).

The figure below shows a simplified example of how the Grid-of-Dots Calibration tool works.



Grid-of-dots calibration

The Grid-of-Dots Calibration tool constructs a transformation object that transforms points between image space and the physical space of the calibration plate.

Once you have created such a calibration, you can use it to transform information returned by vision tools into precise physical locations. Also, since you can associate a transformation with an image supplied as input to a vision tool, you can obtain vision tool results directly in terms of physical locations.

The Grid-of-Dots Calibration tool can construct two types of transformation objects:

- A linear transformation object defined by a (2x2) matrix and a (2x1) vector (see *Math Foundations of Transforms* in the *CVL User's Guide* for more information on linear and nonlinear transformations in CVL).
- A nonlinear transformation object defined by a polynomial transformation (see *Math Foundations of Transforms* in the *CVL User's Guide* for more information on nonlinear polynomial transformations in CVL).

Nonlinear transformation provides higher levels of mapping accuracy but typically takes longer to execute.

Feature Correspondence Calibration

Feature correspondence calibration produces a nonlinear transform derived from either acquisition system parameters, or from a set of corresponding feature points. A feature point is the location of a feature in image space and its corresponding location in physical space. These two methods are discussed below.

Using Acquisition System Parameters

Acquisition system parameters are either intrinsic or extrinsic. Intrinsic parameters are parameters internal to the camera and include:

- The x- and y-scale (also called focal lengths)
- Skew
- The x- and y-translation
- Coefficient of radial distortion

If a camera is calibrated accurately, these parameters remain constant as long as the internal camera settings (camera itself, lens used on the camera, focal length setting, aperture setting, and so on) remain constant. Also, the intrinsic camera calibration parameters are expected to stay constant even if external settings vary such as orientation of object plane with respect to the camera, lighting conditions, and so on.

Extrinsic parameters are properties of the image acquisition system that are external to the camera. These include:

- Orientation of the physical coordinate system with respect to the camera coordinate system, specified by three angles.
- Translation of the origin of the 3D physical coordinate system with respect to the camera coordinate system.

Using Feature Correspondence

A feature correspondence consists of two points: the location of a feature in image space and its corresponding location in physical space. Typically, these features are marks on a calibration plate such as a dot on a grid-of-dots calibration plate, or a vertex on a checkerboard calibration plate. The Feature Correspondence Calibration tool is purposely general in this regard to dissociate itself from any specific calibration plate.

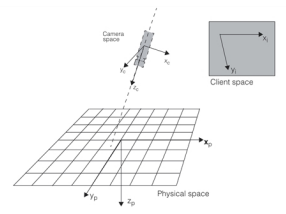
In general, extraction of image feature points and their correspondence with physical points can be performed by your own software tools. You do not have to use a calibration plate image. The essential requirement is that image feature point extraction be accurate and the correspondence of image feature points with physical points be correct.

The Feature Correspondence Calibration tool will work with a single set of calibration plate features extracted from a single image, or if you provide multiple feature sets from multiple images tool accuracy improves. In fact, the tool is designed to work best with features from a number of images each taken at slightly different oblique angles to the same calibration plate.

For your convenience, CVL includes a feature extractor global function (**cfCalib2VertexFeatureExtract()**) designed to extract features from a checkerboard calibration plate and to build the feature sets needed to create feature correspondence calibration objects. If you wish to use a different calibration plate such as a grid-of-dots calibration plate or some other calibration environment, you will need to write your own feature extractor software.

Coordinate Systems

The three coordinate systems used in calibration are depicted in the figure below.



Calibration coordinate systems

x_c, y_c, z_c is the camera coordinate system and x_p, y_p, z_p is the physical coordinate system. Both are 3D, right-handed, rectangular coordinate systems. These coordinate systems do not have skew or aspect. The z_c axis of the camera coordinate system points towards the object viewed by the camera. The physical space z_p axis points away from the camera. The transformation between physical space and camera space is a 3D rigid transformation composed of rotation and translation.

For 2D camera calibration in which a checkerboard calibration plate is used to provide the feature correspondence data, the 2D physical coordinate system defined by x_p and y_p corresponds to the Plate2D coordinate system. The 3D physical coordinate system in this case is the extension of Plate2D with the z_p axis.

Client space is a 2D coordinate system. Client coordinates are transformed by the client coordinate transform that is part of the pel buffer. For many applications this client coordinate transform is an identity transform, in which case image coordinates are the same as client coordinates. These coordinates may contain skew and aspect.

How the Calibration Tools Work

This section describes how the Calibration tools work.

Grid-of-Dots Calibration

The following sections describe how to create calibration objects directly from a grid-of-dots calibration plate.

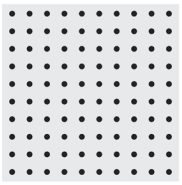
The Grid-of-Dots Calibration Plate

The Calibration tool works by acquiring an image of a calibration plate that you provide. The calibration plate provides the physical reference for the calibration.

The calibration plate you construct must meet the following requirements:

1. It must consist of a two-dimensional grid of dots.
2. The dots must be circular.
3. The diameter of the dots must be no greater than one-half of the smaller of the x-axis dot pitch or the y-axis dot pitch.
4. The diameter of the dots must be at least 10 pixels.
5. The edge of the calibration plate should be outside of the acquired image.

The figure below shows an example of a grid-of-dots calibration plate suitable for use with the Grid-of-Dots Calibration tool.



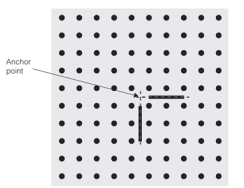
Grid-of-dots calibration plate

You can optionally include a pair of rectangles that meet the following requirements:

1. The length of the rectangles must be 2 times the grid pitch in the direction of the rectangle.
2. The rectangles must be perpendicular to each other.
3. The intersection of the two lines drawn through the centers of the two rectangles must fall at the center of a grid cell.

The Grid-of-Dots Calibration tool lets you use the point defined by the intersection of the lines drawn through the two rectangles as an anchor point for the physical coordinate system.

The figure below shows an example of a calibration plate that includes the two rectangles. (The dotted lines in the figure below are *not* part of the plate but are drawn to show the location of the anchor point.)



Grid-of-dots calibration plate with anchor point

Establishing Translation

By default, when you use the Grid-of-Dots Calibration tool, the tool creates a transformation in which the origin of the image coordinate system maps to the origin of the client coordinate system. You can optionally specify a fixed point within the coordinate plate, usually the anchor point on the calibration plate, as the origin point for the client coordinate system. In either case, this establishes the translation component of the transformation.

Analyzing the Calibration Plate

The Grid-of-Dots Calibration tool works by determining the precise locations of each dot on the plate, then computing the transformation based on the x-axis and y-axis dot pitch that you specify.

The Grid-of-Dots Calibration tool uses the following steps to determine the dot locations:

1. The input image is segmented into background and object pixels. The tool performs this segmentation automatically.

   The tool uses a soft threshold segmentation to avoid spatial quantization error.

2. The Grid-of-Dots Calibration tool applies a vision tool to the segmented image to determine the centers of mass of each object in the image. The center of mass is determined with sub-pixel accuracy.
3. If the tool detects the presence of two rectangles, it computes and stores the location of their intersection point, as shown in *Grid-of-dots calibration plate with anchor point* on page 163.
4. You program the Grid-of-Dots Calibration tool to compute the linear or nonlinear polynomial transformation that best fits the points (see *Math Foundations of Transforms* in the *CVL User's Guide* for more information on linear and nonlinear transformations in CVL).
5. The tool computes the residual error for each point on the grid.

Residual Error

After constructing the transformation object, the Grid-of-Dots Calibration tool measures the distance between pixel locations in the image and corresponding locations predicted by the transformation object. If the tool constructed a linear mapping between image and client coordinates, the *residual error* measures the uncorrected error remaining in your application, both linear and nonlinear. If the tool maps image and client coordinates by using a nonlinear polynomial transformation, the residual error measures how much of the nonlinear distortion is not corrected. If you need higher levels of mapping accuracy you may need to increase the order of the polynomial mapping (see *Math Foundations of Transforms* in the *CVL User's Guide*).

Diagnostic Information

The Grid-of-Dots Calibration tool provides the following diagnostic information:

- An edge-detected image that shows all the high-contrast edges detected by the tool
- The segmented image
- The segmentation threshold computed for the image
- The centers of mass of all of the dots used to compute the calibrated transformation

You can use this information to verify that the tool is working correctly.

Feature Correspondence Calibration

You typically build a calibration transform from a series of corresponding image points and physical points. These points are called features and are extracted from calibration plate images by special software tools. Cognex currently provides **cfCalib2VertexFeatureExtract()**, a global function for extracting features from checkerboard calibration plates. **cfCalib2VertexFeatureExtract()** is covered in *Using Feature Correspondence* on page 165.

You can also create a calibration transform by providing intrinsic and extrinsic parameters from your application environment. Creating a transform from known parameters is covered in the next section.

Using Known Parameters

Use of known parameters as described in this section requires detailed knowledge of your application, your equipment, and its calibration requirements. The parameters required to create a calibration transform are categorized as extrinsic parameters and intrinsic parameters as described below.

Extrinsic Parameters

Extrinsic parameters are properties of an image acquisition system that are external to the camera. Extrinsic parameters include:

- Orientation of the client coordinate system with respect to the camera coordinate system, specified by three angles. We use the convention of *Givens* rotations (see *Multiple View Geometry in Computer Vision*, Appendix A3.1.1, by *Hartley & Zisserman*).

  A rotation R of a 3D coordinate frame is expressed as:

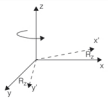
  $R = R_x * R_y * R_z$

  where

  R_z = rotation of x,y-axes about a fixed z-axis,
  R_y = rotation of z,x-axes about a fixed y-axis,
  R_x = rotation of y,z-axes about a fixed x-axis.

  The order of applying rotations is: R_z first, R_y second, and R_x third.

  For example, the figure below shows R_z where the x,y frame is rotated R_z degrees about the z-axis in the x-y plane, to x',y'.

  Rz example

- The translation of the origin of the 3D physical coordinate system with respect to the image coordinate system. This is simply a **cc3Vect** expressing the displacement of the origin in the x, y, and z directions.

Intrinsic Parameters

Intrinsic parameters are properties of an image acquisition system that are internal to the camera. Intrinsic parameters include:

- The x- and y-scale (also called focal lengths)
- Skew
- The x- and y-translation
- Coefficient of radial distortion

The scale, skew, and translation are common **cc2Xform** parameters and are discussed in the *Images and Coordinates* chapter of the *CVL User's Guide*. The coefficient of radial distortion describes both *barrel distortion* and *pincushion*

distortion. Please see the **ccCalib2ParamsIntrinsic***Using Feature Correspondence* on page 165 for a description of these distortion types.

These parameters remain constant as long as the internal camera settings (camera itself, lens used on the camera, focal length setting, aperture setting, and so on) remain constant. Also, the intrinsic camera calibration parameters are expected to stay constant even if external settings vary such as orientation of object plane with respect to the camera, lighting conditions, and so on.

Using Feature Correspondence

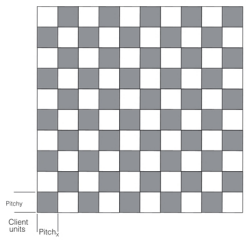
You use feature correspondence to create a calibration transform from a set of corresponding feature points. Each feature point includes the feature location in image space, and its corresponding location in physical space. While you can create a set of corresponding feature points in many ways, you typically create the points from an image of a calibration plate. For example, for a grid-of-dots calibration plate feature points are the dot centers. For a checkerboard calibration plate, feature points are the rectangle vertices. In this latter case, the 2D physical space corresponds to the Plate2D coordinate system, as stated previously.

CVL provides a global utility (**cfCalib2VertexFeatureExtract()**), which you can use to create a vector of feature points from an image of a checkerboard calibration plate. This utility is described below. If this utility does not satisfy your requirements you can write a similar utility of your own.

Using a Checkerboard Calibration Plate

Using images of a calibration plate taken from your application is the most common way to create a feature correspondence transform. It is empirical in that it measures the actual environment where your application runs.

An example of a calibration plate used with **cfCalib2VertexFeatureExtract()** is shown in the figure below. It is a checkerboard of black and white rectangles where each rectangle is the same size.



Example checkerboard calibration plate

A checkerboard calibration plate is a precision manufactured article where all rectangles are the same size to within tight tolerances. You can purchase a checkerboard calibration plate from Cognex or from other calibration plate vendors. If you purchase a third party calibration plate make sure it conforms to the Cognex specification included in *Checkerboard Calibration Plates* on page 171.

To create a feature correspondence transform using a checkerboard calibration plate you perform the following steps:

1. Acquire calibration plate images (discussed in *Calibration Plate Images* on page 166 below).
2. Run the feature extractor function (**cfCalib2VertexFeatureExtract()**) on each image to extract the locations of each vertex, the points in the image where the corners of four squares meet. The feature extractor creates a vector of correspondences between feature locations in image space (Client2D) and feature locations in physical space (Plate2D).

   When you run the feature extractor, you specify the checkerboard square size in physical units (pitch). This establishes the relationship between pixels and physical units. You also specify an origin for the checkerboard grid. The location of feature points will be reported relative to this origin. Specifying the calibration plate origin is discussed in *Setting the Origin* on page 167. You can also use a checkerboard calibration plate with Data Matrix codes that encode the pitch, in addition to specifying the origin for the checkerboard grid (origin of Plate2D).
3. Use the correspondences from the feature extractions to create a calibration transform of type **cc2XformCalib2**.

Calibration Plate Images

Features extracted from calibration plate images are used to create a **cc2XformCalib2** transform. You can create a calibration transform using a single image but your accuracy will improve if you use multiple images. When you use a single image you cannot compute the intrinsic and extrinsic parameters from the calibration object. The object is however, still useful for correcting distortion in images; for example, with the Image Warp tool. (See *Image Warp Tool* on page 126).

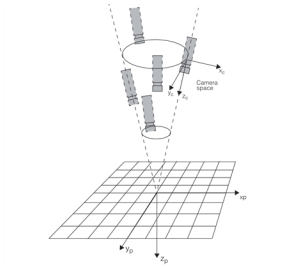
When you use multiple views, the calibration transform computes intrinsic and extrinsic parameters, and accuracy improves. Adding more views increases the accuracy.

Your choice of images will also affect the transform accuracy.

Multiple views of a calibration plate must satisfy the following important restriction:

The planes of the calibration plate relative to the camera cannot be parallel in any two views.

Example camera setup for calibration plate images on page 166 shows an example of a calibration using five views.



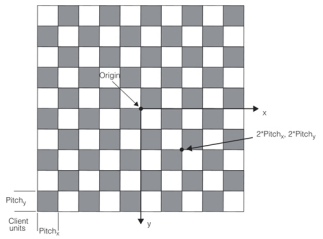
Example camera setup for calibration plate images

Obviously, the same images can be captured using a stationary camera by changing the position and orientation of the calibration plate relative to a fixed camera.

Extracted Features

Run the feature extractor function (**cfCalib2VertexFeaturesExtract()**) on each image. The function locates *vertices* in the image, points where the corners of four checkerboard squares meet. The points found are called *vertex features*.

The function produces a vector of point pairs where each point pair specifies the location of the source image vertex feature in client coordinates, and the feature location in physical units relative to the checkerboard origin. For example, see the figure below.



Extracted features

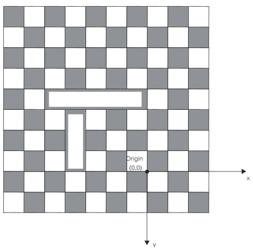
Note that the features extracted from the source image are not described in image coordinates, but are transformed to client coordinates through the source image client coordinate transform. This differs from the grid-of dots calibration where source image locations are described in image coordinates.

$Pitch_x$ and $Pitch_y$ determine the physical location of a feature relative to the origin. In the general case, a feature location is (n*$Pitch_x$, m*$Pitch_y$), where n is the number of checkerboard rectangles from the origin along the x-axis, and m is the similar number along the y-axis. Both n and m can be positive or negative numbers. In the figure above if $Pitch_x$=2 and $Pitch_y$=2, the feature shown is at location (4, 4).

Setting the Origin

You set the checkerboard origin with a run-time parameter you pass to the vertex extractor function. The origin can be set three different ways:

1. Default.
   The feature extractor uses the vertex closest to the image center.
2. Specified point.
   You specify a point in client coordinates and the feature extractor will use the closest vertex as the origin.
3. Fiducial mark.
   If you are using a calibration plate that includes a fiducial mark for establishing the origin. (For example, a Cognex calibration plate). The figure below shows an example of a calibration plate fiducial mark and the associated origin.



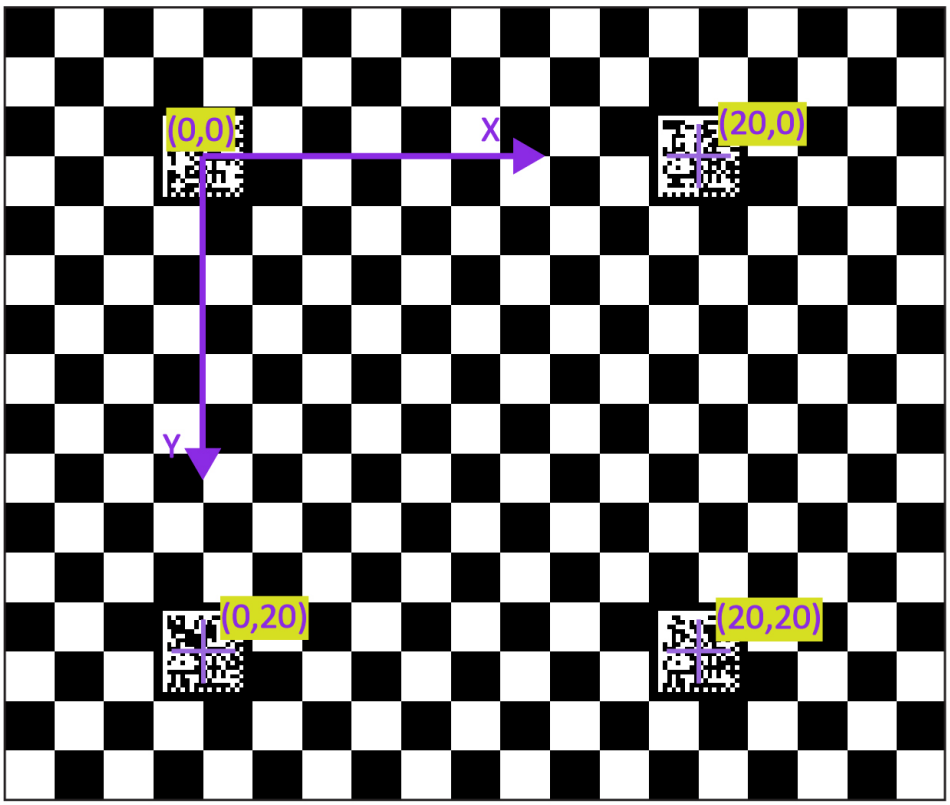
Calibration plate with fiducial mark

You can also use a checkerboard calibration plate in which Data Matrix codes define the checkerboard calibration plate's coordinate space. This calibration plate type is especially useful when:

- A portion of the calibration plate is obscured due to physical obstructions or poor lighting causing the checkerboard grid to get split into distinct regions of contiguous features.
  When Data Matrix codes are used as fiducials, it becomes possible to extract features from these multiple regions as long as each region of contiguous features has a Data Matrix code embedded inside. This is possible because each Data Matrix code encodes its position in the grid.
- A multi-camera system is calibrated to the calibration plate's coordinate space in which system each camera's image is acquired about a (distinct) region of the calibration plate. In this case, having multiple Data Matrix codes that define the coordinate space spread across the calibration plate allows each camera to see some of the Data Matrix codes and thereby the coordinate space.

The following figure shows a checkerboard calibration plate with Data Matrix codes as the fiducial. It also shows the positions of the Data Matrix codes encoded in them and the coordinate space (Plate2D space) the codes define.

Plate2D space – plate with data matrix codes



Calibration plate with Data Matrix codes as the fiducial mark

Using the Calibration Tools

This section describes the procedures, C++ classes, and the global functions you use to create calibration transform objects.

Grid-of-Dots Calibration

The table below summarizes the classes and global functions associated with grid-of-dots calibration objects.

Classes & functions	
cfCalibrationRun()	Global function that creates the calibration transform.
ccGridCalibParams	Run-time parameters.
ccGridCalibResults	Where to place the results.
ccCalibDefs	Enums.
cc2XformLinear	The linear transform returned by the tool when you request a linear calibration.
cc2XformPoly	The nonlinear transform returned by the tool when you request a nonlinear calibration.

Grid-of-dots classes and functions

To use the Grid-of-Dots Calibration tool, follow these steps:

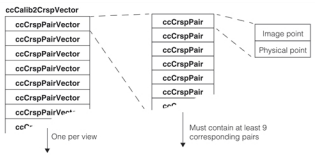
1. Construct or acquire a calibration plate that is of the correct size and orientation for your application setup. See *Checkerboard Calibration Plates* on page 171.
2. (Optional) Include a pair of rectangles on the calibration plate that identify the origin of the physical space or a known offset from the origin of the physical space.
3. Acquire an image of the calibration plate using the optical and physical setup you wish to calibrate. The image should have good contrast and you should make sure there are no shadows, reflections, or other image defects that might confuse the tool.
4. Run the Grid-of-Dots Calibration tool (call **cfCalibrationRun()**).
5. Verify the results of the tool by determining that the expected number of dots were located. (Inspect **ccGridCalibResults**).
6. (Optional) Examine the residual error identified by the Grid-of-Dots Calibration tool to determine if the transformation object returned by the tool provides a sufficiently accurate mapping for your needs. (See **ccGridCalibResults::residuals()**).
7. Obtain the calibration transform (**cc2XformLinear** or **cc2XformPoly**) from the tool result object and store it for use in your application.

Feature Correspondence Calibration

The table below summarizes the classes and global functions that are associated with the feature correspondence calibration.

Classes & Functions	Description
cc2XformCalib2	The calibration transform class.
ccCalib2ParamsExtrinsic	Calibration extrinsic parameters.
ccCalib2ParamsIntrinsic	Calibration intrinsic parameters.
cc3AngleVect	3D coordinate system orientation.
ccCalib2VertexFeatureDefs	Feature extractor defs.
ccCalib2VertexFeatureParams	Feature extractor run-time parameters.
ccCalib2VertexFeatureParseResult	The result of parsing the string that was encoded in a Data Matrix code.
ccCalib2VertexFeatureResult	The result of the calibration.
ccCalib2VertexFeatureSymbolInfo	The data decoded from the Data Matrix code.
cfCalib2VertexFeatureExtract()	Feature extractor global function.
ccCalib2CrspVector	A vector of correspondences for one image. See the figure below. **ccCalib2CrspVector** is a typedef as follows: `typedef cmStd vector<ccCrspPairVector> ccCalib2CrspVector;`
ccCrspPairVector	A vector of corresponding point pairs. See the figure below. **ccCrspPairVector** is a typedef as follows: `typedef cmStd vector<ccCrspPair> ccCrspPairVector;`
ccCrspPairWeightedVector	A weighted vector of corresponding point pairs. This is used with 3D camera calibration.
ccCrspPair	An image point and a corresponding physical point. **ccCrspPair** is a typedef as follows: `typedef ccPair<cc2Vect> ccCrspPair;`

Class and global function summary



Feature correspondence classes

For 2D camera calibration in which a checkerboard calibration plate is used to provide the feature correspondence data, the position of the image point is expressed in the Client2D coordinate system and the position of the physical point is expressed in the Plate2D coordinate system.

You can create a feature correspondence calibration object using parameters, or from sets of feature correspondences. Both procedures are outlined below.

Using Parameters

1. Construct **ccCalib2ParamsExtrinsic** and **ccCalib2ParamsIntrinsic** parameters objects.
2. Create a default-constructed calibration object, **cc2XformCalib2**.
3. Call **cc2XformCalib2::init()** passing the parameter objects from Step 1.
4. The calibration object is ready for use.

Using a Checkerboard Calibration Plate

Note: This procedure is for checkerboard calibration plates that can be used with the CVL feature extractor **cfCalib2VertexFeatureExtract()**. See *Checkerboard Calibration Plates* on page 171.

1. Acquire a checkerboard calibration plate that meets the Cognex specifications. (See *Checkerboard calibration specifications* on page 173). You can use any physical grid pitch as long as the checker size in your images is at least 15x15 pixels. Note that 15x15 pixels is the extreme low end of the operating range and for most applications the checker size should be larger so you are not operating on the range limit.
2. Acquire several images of the calibration plate using the optical and physical setup you wish to calibrate. (See *Calibration Plate Images* on page 166). The images should have good contrast and you should make sure there are no shadows, reflections, or other image defects that might confuse the tool. For 3D camera calibration, the relative orientation of the calibration plate with respect to the camera must be different for each image. When using multiple images, at least two images should be acquired. Accuracy generally improves with each image you add providing your feature extractor is working well. If the feature extractor does not accurately extract vertices from the images, adding images may actually make accuracy worse.
3. Run the feature extractor, **cfCalib2VertexFeatureExtract()**, on each image to create the **ccCrspPairVector** result vector.

   Note: Cognex recommends specifying the (non-default) *eExhaustive* algorithm in the **ccCalib2VertexFeatureParams** that you supply to this function.

   When Data Matrix fiducials are employed, you can either use the *eExhaustive* algorithm or the *eExhaustiveMultiRegion* algorithm. *eExhaustive* only returns features from one fully connected region, but it is faster. *eExhaustiveMultiRegion* can return features from multiple distinct contiguous regions and is therefore more robust, but it is slower. A use case for this is when a portion of the calibration target is not visible (for example, due to physical obstructions or poor lighting), causing the checkerboard grid to get split into distinct regions of contiguous features. For features to be extractable from a region, the region has to have a Data Matrix code embedded inside. This is because each Data Matrix code encodes its position in the grid, which position information is needed for feature extraction. Therefore, *eExhaustiveMultiRegion* can only be used with Data Matrix fiducials.
4. Place all the **ccCrspPairVector** result vectors into one **ccCalib2CrspVector** vector.

5. Create a default-constructed calibration object, **cc2XformCalib2**.
6. Call **cc2XformCalib2::init()** passing the **ccCalib2CrspVector** object from Step 4.
7. The calibration object is ready for use.

> **Note**:
> If you use a single view of the calibration plate, there are no intrinsic or extrinsic parameters computed for the calibration transform.
>
> If you use multiple views, intrinsic and extrinsic parameters are computed. Intrinsic parameters contained in the initialized transform depend on the camera settings. Extrinsic parameters contained in the initialized transform correspond to the first view. (The first **ccCrspPairVector** in the **ccCalib2CrspVector** vector).
>
> If you wish to use extrinsic parameters from a different view, you must call **cc2XformCalib2::extrinsicParams(***view***)** for the desired view before using the calibration object.

If you are not able to obtain a high quality checkerboard calibration plate the procedure above may not produce good results because the physical locations of vertices produced by the feature extractor in Step 3 above will be inaccurate. You can still use the calibration plate to create an accurate calibration by measuring the vertex locations on the calibration plate manually. You can do this with an x-y positioning microscope or equivalent precision device. You would then replace all the physical locations in the **ccCrspPairVector** result vector with the measured values and complete the procedure.

Checkerboard Calibration Plates

If you use the Feature Correspondence Calibration tool you may wish to use the calibration plate feature extractor provided by Cognex to create the feature correspondence list. The feature corresponder is discussed in *Feature Correspondence Calibration* on page 160. The Cognex feature extractor is designed to use images from a checkerboard calibration plate described in this section. You can use specifications included here to build your own checkerboard calibration plate, or you can use CAD files provided by Cognex to have a calibration plate vendor make one for you. Consult your CVL release notes for the location of these CAD files on your release media.

The following section describes the checkerboard calibration plate specifications, and the next section describes some calibration plate image requirements.

Calibration Plate Specifications

The calibration plate must meet the following specifications. See *Checkerboard calibration specifications* on page 173, the bottom figure under Calibration Plate Image Requirements.

1. The calibration plate tiles should be perfectly square. However, the feature extractor that works on the calibration plate image will tolerate tile aspect ratios of from 0.8 to 1.2.
2. The dimensions of the black tiles and the white tiles must be equal.
3. Choose the calibration plate surface material to minimize reflections and non-uniformities. The surface material choice will depend on the camera optics and the surface illumination.
4. Including a registration fiducial in your calibration plate is optional. If you include a fiducial, only use one that conforms to the pattern and dimensions shown in *Checkerboard calibration specifications* on page 173.
5. Calibration plate dimension tolerances must be carefully controlled during the manufacturing process. This is usually more difficult when you make smaller calibration plates. Measure your completed calibration plate carefully to ensure the accuracy of the vertices.

Note:
If you are not able to maintain the needed vertex accuracy on completed calibration plates, the plates can still be used to create a feature correspondence list for calibration. See *Using a Checkerboard Calibration Plate* on page 170.

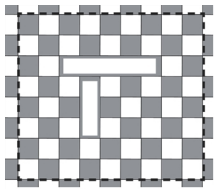
Calibration Plate Image Requirements

The physical dimensions (grid pitch) of the calibration plate must be chosen such that in images acquired through your camera and optics, checkers are no smaller than 15 x 15 pixels. Note that 15x15 pixels is the extreme low end of the operating range and for most applications the checker size should be larger so you are not operating on the range limit.

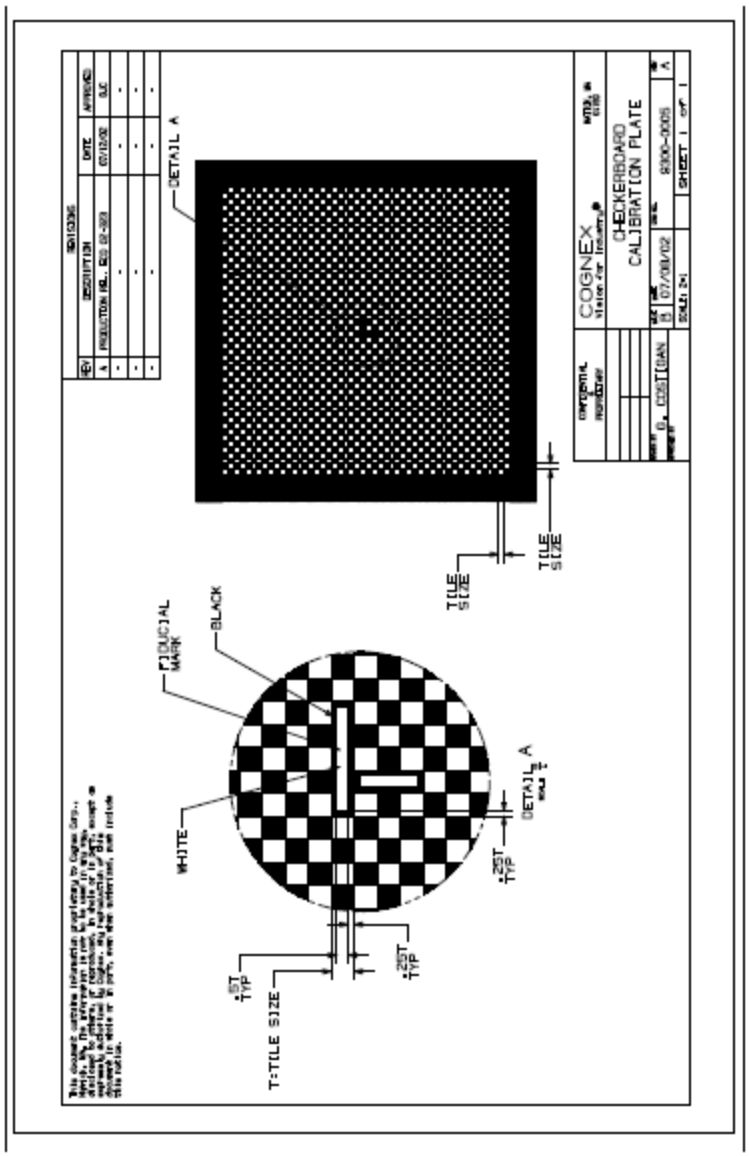
If your calibration plate does not include a fiducial mark, images must contain at least a grid of 3x3 checker vertices. See the figure below.

Example image containing a 3x3 grid of checker vertices

If your calibration plate does have a fiducial mark, the image must contain at least two rows and columns surrounding the fiducial mark. The dotted line box in the figure below outlines the minimum image size.



Example of fiducial image requirement



Checkerboard calibration specifications

Data Matrix Code Requirements for Calibration Plates with Data Matrix Code Fiducials

The requirements for Data Matrix codes used as the fiducial on a calibration plate are as follows:

- Only the ECC200 type Data Matrix codes are supported.
- Each Data Matrix code must be printed/rendered with square modules, with 100% occupancy. The Data Matrix should have a resolution of at least 4 pixels per module (ppm), and no greater than 20 ppm.
- The X and Y axes of all Data Matrix codes in the image should be aligned with the checkerboard tile grid. The X axes of all codes should be parallel to each other, and the Y axes of all codes should be parallel to each other.
- The X and Y axes of the Data Matrix codes define the X and Y axes of the checkerboard coordinate system. The origin of the checkerboard coordinate system falls on the lattice grid of the checkerboard tile corners. The origin has a grid coordinate of (0,0) regardless of whether it is explicitly marked by a code.

- Each Data Matrix code should occupy and be centered on a contiguous region of checkerboard tiles. The region must cover an even number of tiles along the rows and columns.
- The Data Matrix code's center should nominally rest on the lattice grid of tile corners.
- Each Data Matrix code should have a “quiet-zone” of at least one module on all four sides. The quiet-zone should fill up the rest of the checkerboard tiles which the code occupies.
- Each Data Matrix code should encode its data in ASCII. The data must be specified in one of the following formats:

1. Version 0 Format (as a regular expression):

   (\+?|-)\d+,(\+?|-)\d+

   a. Each Data Matrix code should encode the (X,Y) Grid2D position of the Data Matrix center on the lattice grid of the checkerboard tile corners. Note, that the Grid2D position should be specified in the unit of tiles, not physical length units or fiducial indices.

   b. There are no parentheses or braces around the values.

   c. The X and Y coordinates are separated by a single comma.

   d. Each of the X and Y coordinates is encoded as a signed decimal integer, without any decimal point and with an optional plus sign for positive values. The minus sign is mandatory for a negative value.

   e. There are no spaces before, after, or between the values.

      The following are some examples of valid Version 0 strings:

      50,25

      50,-120

      +50,-45

      -50,+40

      The following table shows the (X,Y) value range available for 14x14 and 16x16 codes that only encode grid-position, using a balanced range for X and Y values with signed coordinates. Note that these are just examples; the tool itself does not enforce any constraint on the size of values used for X or Y coordinates. See <*ch_cvl/acusymbl.h*> for information on supported code sizes.

Code Size	Alphanumeric Capacity	X (or Y) # digits	X (or Y) value range
14x14	10	(10-3)/2 = 3	-999 to 999
16x16	16	(16-3)/2 = 6	-999999 to 999999

      (X,Y) value range available for 14x14 and 16x16 codes that only encode grid-position, using a balanced range for X and Y values with signed coordinates

2. Version 1 Format (as a regular expression):

   (\+?|-)\d+,(\+?|-)\d+ V1 P\d+\.?\d* (MM|IN|MIL)

   a. Similar to the Version 0 format, each Data Matrix code should encode the (X,Y) grid position of the Data Matrix center on the lattice grid of the checkerboard tile corners. See the requirements for the Version 0 format above.

   b. Each Data Matrix code must additionally encode the following:

      -> a version-number (see item c),

      -> the grid-pitch, (see item d), **and**

      -> the measurement units (see item e)

   c. The version number must start with a single space (ASCII 0x20), and it must be followed by the string V1 (as of this release).

   d. The grid pitch must start with a single space (ASCII 0x20), followed by an uppercase P. It must be followed by:

      -> One or more decimal digits

      -> Followed by an optional decimal point (ASCII 0x2E)

      -> Followed by zero or more decimal places.

   e. The measurement units portion must start with a single space (ASCII 0x20), followed by exactly MM or IN or MIL, which correspond to millimeters, inches, and thousandths of an inch respectively.

For example, a Version 1 string that encodes a 0.5mm grid-pitch for the Data Matrix code centered at grid position (123,456) will look like:

123,456 V1 P0.5 MM

Note that this format makes the “0” mandatory before the decimal point.

The Version 1 format might require the use of Data Matrix codes larger than 14x14 or 16x16. The following table lists some common code sizes and their corresponding alphanumeric capacity.

Code Size	Alphanumeric Capacity
14x14	10
16x16	16
18x18	25
20x20	31
22x22	43

Common code sizes and their corresponding alphanumeric capacity

- The Data Matrix code can be square or rectangular, and it is okay to mix Data Matrix codes of different sizes in the same calibration target. See <*ch_cvl/acusymbl.h*> for information on supported code sizes.

- The tool supports auto-detection of polarity, size and mirroredness.
    - The Data Matrix codes can be placed arbitrarily on the checkerboard, subject to the following constraints:
    - The distance between any two codes (from edge to edge) should be at least 4 checkerboard tiles.
    - The Data Matrix codes themselves do not need to form a regular grid, as long as all of them are centered on the lattice grid of checkerboard tile corners, and have correct code data to reflect their grid positions.
    - Each image should have at least one decodable and parseable Data Matrix code. A Data Matrix code is considered parseable if it adheres to the format described previously (Version 0 Format and Version 1 Format).
    - The tool requires that Data Matrix codes be embedded within a checkerboard pattern. The Data Matrix codes must be connected to the checkerboard tiles - codes that are not connected to checkerboard tiles will be ignored.

Line Scan Camera Calibration

This chapter describes the Line Scan Camera Calibration tool, a tool that allows you to perform nonlinear calibration upon line scan cameras.

This section and *Some Useful Definitions on page 177* give an overview of the chapter and define some terms you will encounter as you read.

The chapter has the following major sections:

Line Scan Camera Calibration Overview on page 177 introduces the Line Scan Camera Calibration tool and describes its purpose.

Using Line Scan Camera Calibration on page 181 describes how to use the Line Scan Camera Calibration tool.

Differences Between the Line Scan Camera Calibration API and the Area Scan Camera Calibration API on page 183 describes the API differences between the Line Scan Camera Calibration tool and the area scan camera calibration tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

line scan camera: A camera that acquires an image by acquiring a single row of pixels at a time.

transform: A C++ class that can perform mapping between two coordinate systems. Most often used in CVL to map between image space and client (physical) space. Transform mapping can be linear or nonlinear.

Line Scan Camera Calibration Overview

The line scan calibration API determines the line scan acquisition model parameters from a set of <image, physical> correspondences from a single image.

Linescan camera calibration, like area-scan camera calibration, is performed by acquiring an image of a calibration plate, then calling a camera calibration function to perform the calibration. Linescan camera calibration requires the use of a checkerboad calibration plate that meets the requirements specified in the topic *Checkerboard Calibration Plates on page 171*.

Linescan camera calibration establishes a calibration transformation object (**cc2XformCalib2**, defined in *ch_cvl\ccalib.h*) that maps points from a physical space defined by the planar motion associated with the line scan camera to the image space of the image acquired by the camera. The calibration transformation object is established with lens distortion mode set to *eLineScan.*

You must call either the 1D warper or the 2D warper to create an unwarped version of the image acquired by the camera. The unwarped image removes the distortion associated with camera rotation, camera tilt, optical distortion, and uneven pixel size or spacing.

Coordinate Spaces in Line Scan Camera Calibration

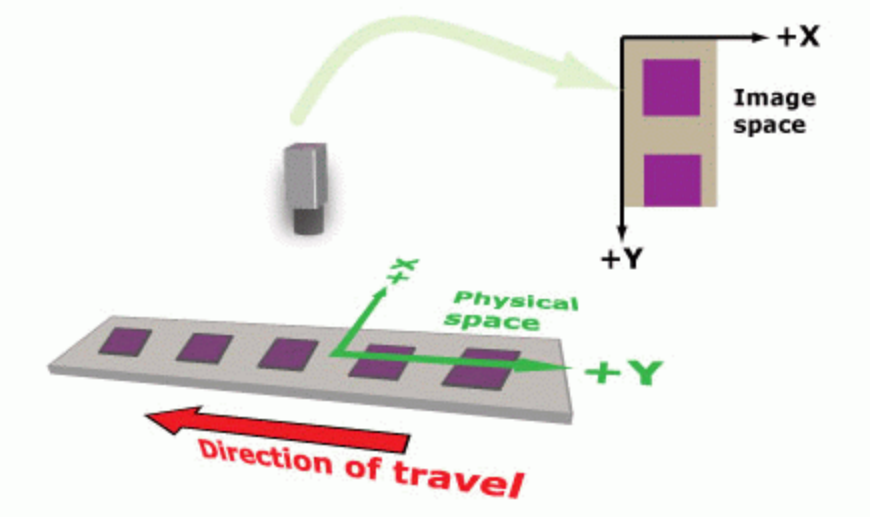
The physical coordinate system for line scan camera calibration is established as follows:

1. The y-axis of physical space is defined by the apparent movement of the line scan camera. The positive y-direction is defined by scan order (the y-value increases in the order in which lines are acquired).
2. The x-axis of physical space is normal to the y-axis. The positive x-direction defines a left-handed coordinate system.

3. The origin point (0,0) of the space is user-defined. In the case where the linear motion is finite, the origin may be at the center of the motion or at either end. In the case where the linear motion is infinite, as with a web, only the x-coordinate is meaningful.
4. The x- and y- units are defined by the grid pitch that you specify during calibration.

The image coordinate system is the same as for other images.

The figure below illustrates the coordinate spaces associated with line scan camera calibration.



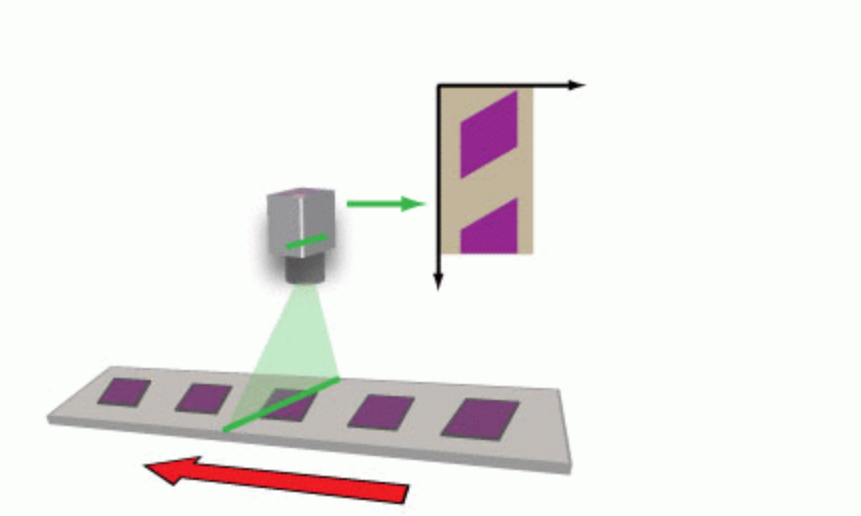
Coordinate spaces associated with line scan camera calibration

Line Scan Acquisition Distortion Correction

Different types of distortion can be present in images from a line scan camera.

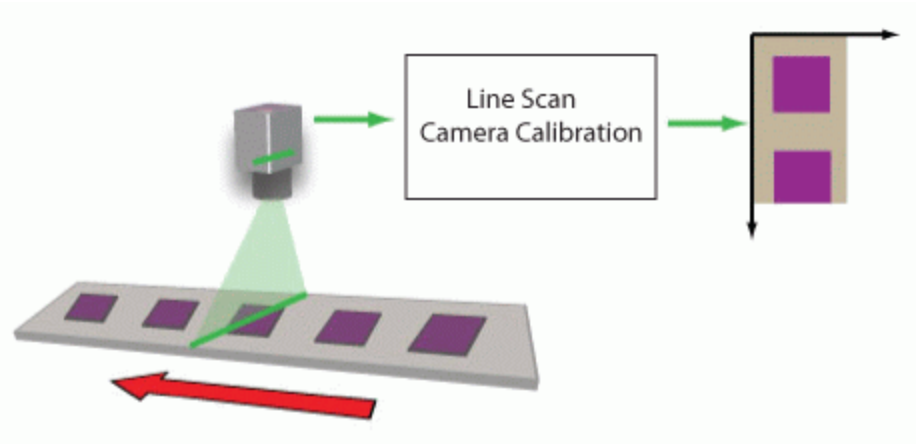
Camera Rotation

If the line scan camera is rotated with respect to the direction of linear motion (the image sensor is not perpendicular to the direction of travel), features in acquired images will be skewed, as shown in the figure below.



Camera rotation and its result

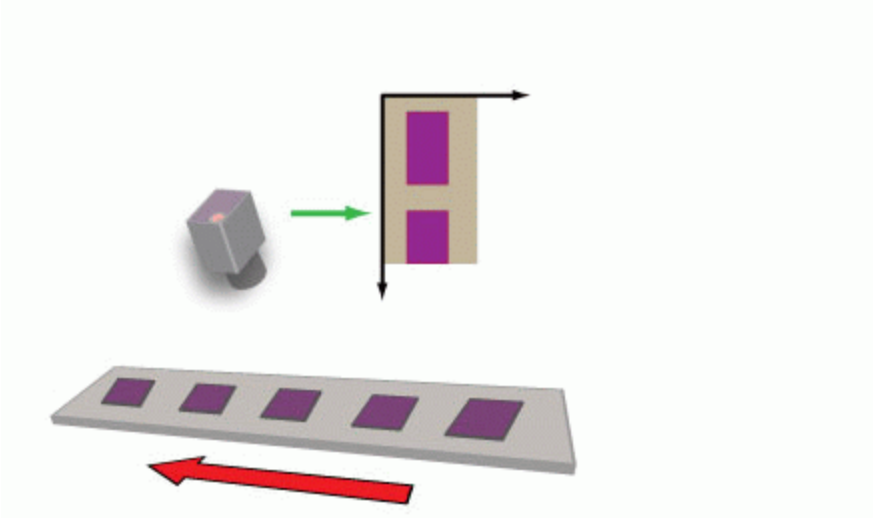
As long as the camera rotation is within ±5°, line scan camera calibration can rectify the distortion as illustrated in the figure below.



Camera rotation correction

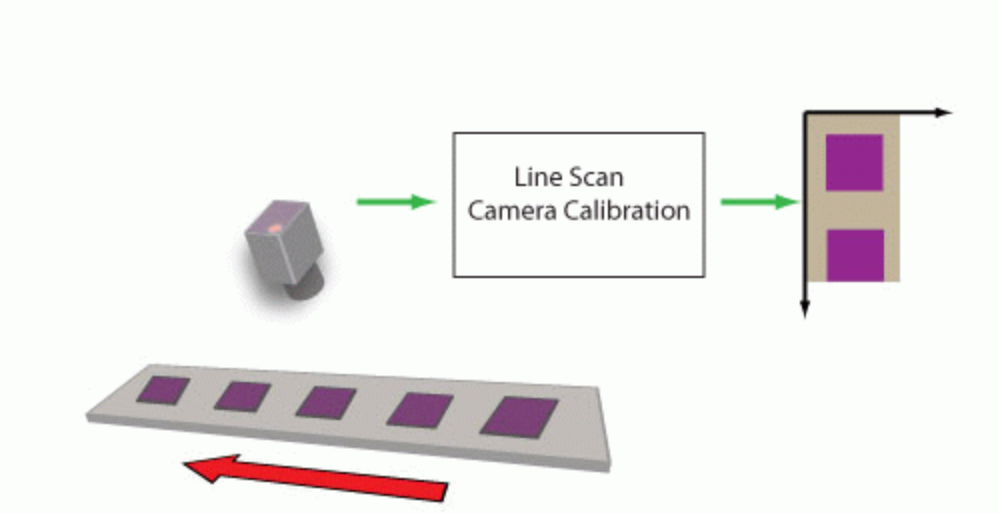
Camera Tilt

If the line scan camera is tilted with respect to the plane of motion (the image sensor is not parallel to the plane of motion), features in acquired images will exhibit a change in their apparent aspect ratio, as shown in the figure below.



Camera tilting and its result

As long as the camera tilt is within ±5°, line scan camera calibration can rectify the distortion as illustrated in the figure below.



Camera tilting correction

Radial Distortion

Images acquired with a line scan camera are subject to the same types of radial distortion as images acquired with area scan cameras. The optical distortion in a line scan image is limited to the x-direction, so rather than the classic barrel distortion or pincushion distortion, the distortion manifests itself as nonlinear distortion across the x-dimension of the image, as shown in the figure below.

Radial distortion

Line scan camera calibration uses the same techniques as the standard nonlinear calibration tool to remove this distortion.

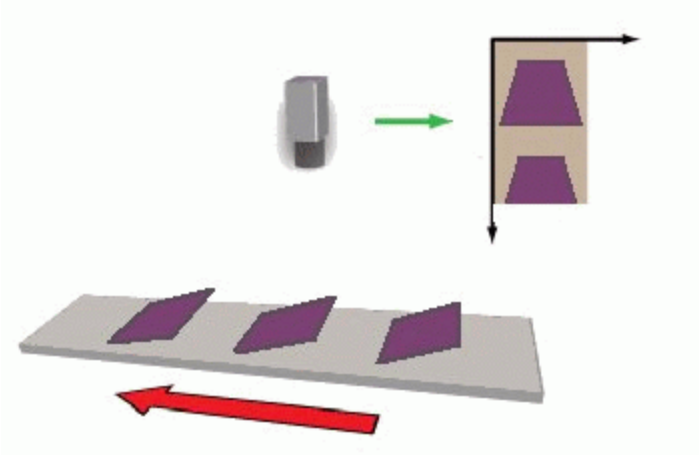
Non-linear Translation Distortion

To the extent that the linear motion is not precisely synchronized with the encoder pulses that drive the acquisition of individual lines, line scan images may exhibit nonlinear distortion in the y-direction. This distortion is not corrected by the linescan camera calibration tool.

2D Warping

The types of distortion previously described, such as camera tilt and camera rotation, can be corrected with line scan calibration and its one-dimensional warping transformations.

For some production environments, 1D transformations might not be sufficient. For example, if the object under inspection can be tilted with respect to the direction of linear motion, features in acquired images will demonstrate some degree of perspective distortion, as shown in the figure below.



Perspective distortion

For these cases, using the 2D image warper (defined in *ch_cvl\imgwarp.h*) is recommended.

Be aware that when you use the 1D warping transformation, the image of your calibration plate needs to capture the width of the plate while the height can be of arbitrary size. When you choose the 2D warping transformation, however, the calibration image must be the same width and height as your run-time images.

Using Line Scan Camera Calibration

This section describes how to perform line scan camera calibration.

The tool is used in two phases. First, an image of the calibration plate is acquired and the calibration is computed. This is called the calibration phase. Once the calibration is computed, you call the 1D or 2D image warper to unwarp the input image. This is called the run-time phase.

You use the **ccCalibrateLineScanCameraParams** class to define calibration parameters and the **cfCalibrateLineScanCamera()** functions to calibrate line scan cameras.

Calibration Phase

To perform line scan camera calibration, follow these steps:

1. Rigidly mount the camera so that its image sensor is perpendicular to the direction of travel and parallel to the plane of travel (within ±5°).
2. Adjust the camera focus so that a sharp image at the plate surface can be acquired.

> **Note**: Moving the camera, changing its lens, focus, or aperture will invalidate the calibration once it has been computed. Make sure the camera is where you want it before proceeding.

3. Affix the calibration plate so that its movement relative to the camera is the same as the parts that are being imaged. If your physical configuration has a meaningful origin and you are using a calibration plate with a fiducial mark, place the mark so that it indicates the desired physical position. The plate should be sufficiently large that the plate fills the width of the image.

   The line scan calibration provides functionality for the automatic alignment of the physical coordinate system with the scanning motion. Line scan calibration parameters include an enum member called **calibrationYAxisAdjustmentMode()** that can be:
   ccCalibrateLineScanCameraDefs::eUsePlateOrientation – the orientation of the plate is used,
   ccCalibrateLineScanCameraDefs::eAdjustOrientationToScanDirection– the computed physical space will be aligned with the scanning motion automatically.

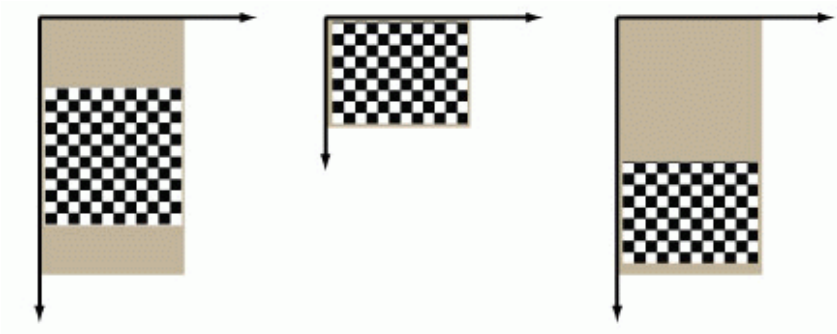
   Note: The automatic alignment functionality provided by **calibrationYAxisAdjustmentMode()** is a prerequisite for using the 1D warper.

4. Accurately measure and record the distance between the surface of the calibration plate and the camera's image sensor (working distance). The accuracy of the measurement must be better than ±10% for the calibration to be accurate. You specify the *distanceFromCameraToTarget* parameter using the parameters class.

5. Acquire an image of the plate.

   If you are using the 1D warping transformation option, the image should be large enough to contain at least ten rows of calibration plate squares but there is no upper limit on the size of the image and there is no requirement that the image be filled in the vertical dimension by the plate. If you are using the 2D warping transformation option, the calibration image must match the height and width of the desired run-time images.

   The figure below shows examples of valid calibration images.



Valid calibration images

6. Run the checkerboard feature detector (specified in *ch_cvl\calibftr.h*), and specify the grid pitch and whether or not the calibration plate includes a fiducial mark.

7. Run the Line Scan Camera Calibration tool, and specify the distance from the camera to the plate in the same units as the grid pitch and the y-axis adjustment mode.

In addition to reporting whether the calibration was successful, the calibration tool also computes and reports the RMS error for the calibration. You can use this information to assess the accuracy of the calibration.

Run-Time Phase

At run time, acquire an image using the same camera and configuration used for calibration, and call the 1D or 2D image warper supplying the acquired image as input to the functions. If you are using the 2D warping option, the images must have the same width and height as your calibration image.

Limitations

The Line Scan Camera Calibration tool has the following limitations:

- The calibration object does not provide access to the line scan parameters (such as the motion vector). The calibration object provides a functionality for mapping between physical space and image space; this mapping incorporates lens distortion, perspective distortion, and the motion vector.

  Note: Because this calibration is from a single-view, the **cc2XformCalib2** object will throw *cc2XformCalib2Defs::BadParams* if you try to get either the intrinsic parameters or the extrinsic parameters.

- The tool expects that the sensor corresponds to the image's x-axis ; that is, each horizontal row of an image corresponds to the responses of the sensor elements. Consequently, do not try to calibrate with a transposed image.
- Use only this tool to perform line scan camera calibration. The **cc2XformCalib2** function does not directly support line scan camera calibration. **cc2XformCailb2.init()** will throw *cc2XformCalib2Defs::NotImplemented* if the distortion model is **cc2XformCalib2Defs::eLineScan**.
- Like all calibration functions, the line scan camera calibration is asymmetric between image positions (which are usually the same as client coordinates) and physical positions. You should always call **cfCalibrateLineScanCamera()** with image positions first, and physical positions second. Similarly, it is important to specify the positions in the **ccCrspPairVector** in the proper order: the **ccCrspPairVector** passed to **cfCalibrateLineScanCamera()** should always be paired with image positions first; that is, (<image position>, <physical position>). Note that client coordinates are usually the same as image coordinates.

Differences Between the Line Scan Camera Calibration API and the Area Scan Camera Calibration API

The API for line scan camera calibration differs from the API for area scan camera calibration in the following ways:

- The line scan API uses a **cfCalibrateLine ScanCamera()** global function and a parameters class, whereas area scan cameras use an **init()** member of the **cc2XformCalib2** class.
- The line scan API only supports single-view camera calibration whereas the area scan camera calibration supports multi-view camera calibration.
- The line scan API allows the user to specify an approximate working distance from the camera. This information can improve the accuracy of the line scan camera calibration.
- The line scan API supports physical coordinate realignment, wherein the physical coordinate system can optionally be defined to be parallel to the scanning motion. This capability enables the use of the 1D warper (*ch_cvl\imgwrp1d.h*) which requires that the transform be linear in the y-direction.

Line Scan Distortion Correction Tool

This chapter describes the Line Scan Distortion Correction tool, a tool that rectifies an image that contains nonlinear distortion in the x-axis direction without the need to use a calibration plate.

This section and *Some Useful Definitions on page 184* give an overview of the chapter and define some terms you will encounter as you read.

The chapter has the following major sections:

Line Scan Distortion Correction Tool Overview on page 184 introduces the Line Scan Distortion Correction tool and describes its purpose.

Using the Line Scan Distortion Correction Tool on page 185describes how to use the Line Scan Distortion Correction tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

line scan camera: A camera that acquires an image by acquiring a single row of pixels at a time.

transform: A C++ class that can perform mapping between two coordinate systems. Most often used in CVL to map between image space and client (physical) space. Transform mapping can be linear or nonlinear.

Line Scan Distortion Correction Tool Overview

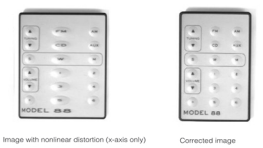
All images contain some distortion, for example, caused by imperfections in camera lenses. In many applications, the distortion does not affect the operation of vision tools, but for applications requiring high accuracy or applications with high distortion, it must be corrected to achieve the desired results.

The Line Scan Distortion Correction tool can measure and correct for nonlinear distortion (for example, lens distortion) in line scan image acquisition systems.

The Line Scan Distortion Correction tool analyzes the pattern in a training image and estimates the distortion.

The Line Scan Distortion Correction tool resamples each row of a run-time source image according to the measured distortion or to a user-provided set of sample positions to generate an undistorted run-time destination image. It resamples the entire field of view in a uniform physical fashion.

The figure below shows the type of distortion that this tool can correct.



Distorted and corrected image

The key differences between calibration tools described in section *Calibration Tools* on page 158 and the Line Scan Distortion Correction tool are the following:

- The Line Scan Distortion Correction tool uses a simple training target that is placed at a fixed position in front of the camera (calibration tools require that a calibration plate be moved relative to the camera).

- The Line Scan Distortion Correction tool does not perform any calibration; no calibrated space is added to the image coordinate space tree (calibration tools compute a calibrated physical coordinate space and add it to the image's coordinate space tree).
- The Line Scan Distortion Correction tool does not rectify distortion caused by a camera that is rotated; it can only correct nonlinear optical and perspective distortion (calibration tools do rectify the image skew caused by a rotated camera).
- The Line Scan Distortion Correction tool estimates 3rd order polynomial to characterize the distortion map.

In general, you should use calibration tools, as they provide the most accurate distortion correction and they provide calibration in addition to distortion correction. The Line Scan Distortion Correction tool is intended to be used in applications where it is impossible to use a moving calibration plate.

Using the Line Scan Distortion Correction Tool

Before you can use the Line Scan Distortion Correction tool to correct distortion in run-time images, you must train the tool. Therefore, the tool has both a training phase and a run-time phase. This section describes how to train the Line Scan Distortion Correction tool and how to use the trained tool to correct nonlinear distortion in run-time images in line scan image acquisition systems.

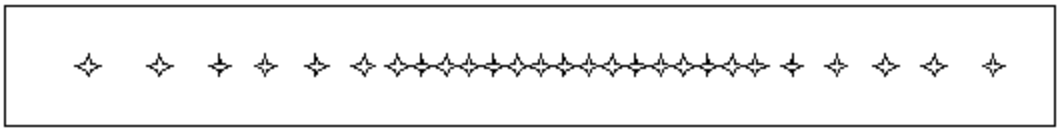
Training the Line Scan Distortion Correction Tool

The Line Scan Distortion Correction tool is trained from either an image of a scene of a regular repeating stripe pattern or a user-provided set of sample positions.

The following figure shows an example of an acquired line scan training image that exhibits significant distortion.

Line scan training image exhibiting significant distortion

The following figure shows the set of sample positions (where the source image is sampled to get the destination image) either calculated based on the training image or provided by the user.



Set of sample positions

The following figure shows the result for the same line scan image undistorted by the Line Scan Distortion Correction tool using the set of sample positions.

Undistorted line scan image result – the destination image

Training the Line Scan Distortion Correction Tool Using an Image

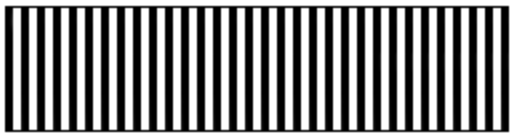
When the tool is trained from an image, the line scan image should be in free running mode and the camera and the scene should be stationary so that the acquired image looks like a series of vertical stripes.

You must train the tool using a training target that meets the following requirements:

1. The training target must be large enough to fill the entire field of view of your camera when it is placed at the working distance used by your application.
   Otherwise, the tool may perform suboptimally in the case where the training region is smaller than the run-time region because the tool will extrapolate the distortion for the uncovered portions of the training image.
2. The surface of the target must have a series of alternating, equally sized white and black regions where:
   - The difference in grey levels in an image of the target must exceed 10.
   - The regions should be sized so that at least 30 sets of black and white regions are visible across the camera's field of view.
   - The black and white regions must be of equal width.
   - In the acquired image of the target, each black and white region must be at least 10 pixels wide.
   - The boundaries between the black and white regions must be parallel.

   Note: This tool expects that the lens distortion only occurs along the x-direction. This tool will perform suboptimally if a transpose image is used or if an image from an area scan camera is used.
3. The training target must be constructed so that it can be rigidly mounted at the working distance of your application such that the points defined by the intersection of a line normal to the direction of travel with the boundaries between dark and light regions are collinear; the target must be flat in the x-direction.

The following figure shows a valid training target.

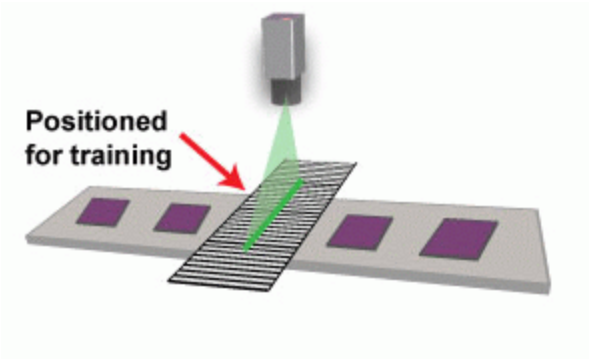


Valid training target

To train the Line Scan Distortion Correction tool, perform the following steps:

1. Mount your camera in the location where it will be used at run time. Adjust your focus, lighting, and aperture to meet the requirements of your application.

   Note: You must not change the physical or optical characteristics of the camera after training.
2. Place the training target such that the camera field of view is filled by the target, the boundaries between the black and white regions are parallel to the direction of travel, and the surface of the target is at the same height as the parts or surface being inspected.
   The following figure shows the positioning of a valid training target.



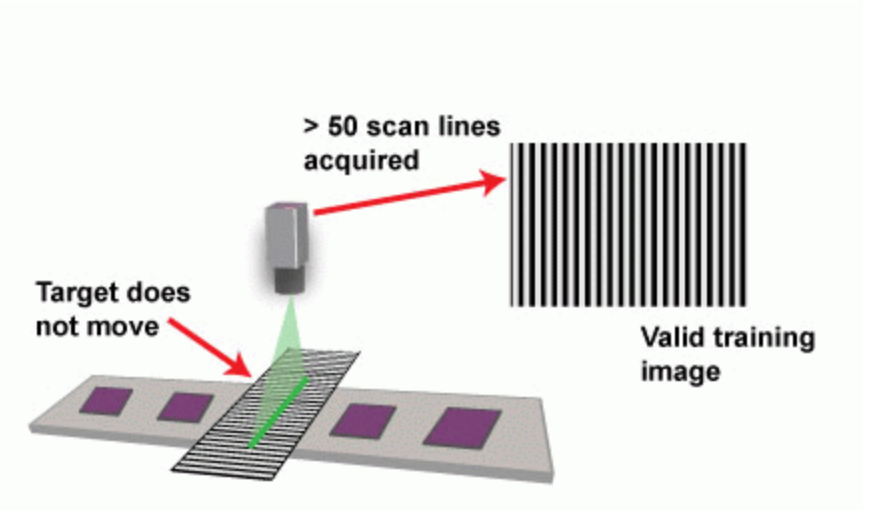
Positioning of a valid training target

3. *Without* moving the target, acquire an image with at least 50 scan lines. The acquired image should appear to contain a series of alternating dark and light stripes.

> **Note**: For information on acquiring an image for training, see the section about acquiring with line scan cameras in the *CVL User's Guide*.

4. Call the **train()** function of the tool.

The following figure provides an overview of the training image acquisition process:



Overview of the training image acquisition process

Training the Line Scan Distortion Correction Tool Using a User-Provided Position Set

In addition to using image-based training, the Line Scan Distortion Correction tool also allows you to train the tool using a set of x-offset positions that you supply, where the offsets are the set of image locations that correspond to a set of evenly spaced physical positions on the plane being inspected.

If you use position-based training, you must determine the image coordinate x-offset positions that correspond to evenly spaced locations in the physical world.

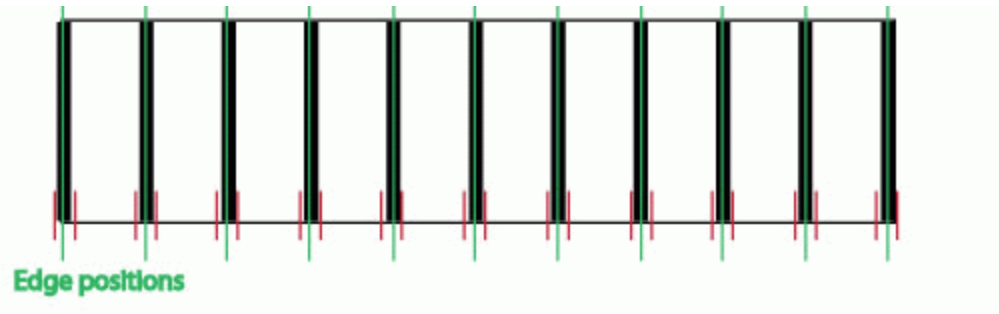
Position-based training may be needed if your application requires the use of a training target that does not conform to the requirements for image-based training targets. For example, suppose that your application used a training target that consisted of equally spaced dark bars on a light background, but the bars were narrower than the light spaces between bars as shown in the following figure.

Invalid training target with narrow bars

While this target does contain evenly spaced features, because the size of the dark bars does not match the size of the light regions between bars, it cannot be used for image-based training requiring equally sized dark and light bars.

To use an image of this target with position-based training, configure a Caliper tool to find edge pairs, where the left edge had a negative (light-to-dark) polarity and the right edge had a positive (dark-to-light) polarity. When configured to

search for edge pairs, the Caliper tool returns the location of the center point of the pair. In this case, the center point corresponds to the center point of the bar.



Edge pairs found with the Caliper tool

Note: Unlike image-based training, there is no requirement to acquire multiple image lines for training. For best results, however, Cognex recommends that you acquire multiple image lines before running the Caliper tool.

To train the Line Scan Distortion Correction Tool using user-provided position-based training, call the tool's **train()** function providing the edge pair position results in order.

Serialization Support

The tool supports serialization in that it can be encoded and decoded from a byte stream.

A trained tool is encoded with its trained state, and when it is decoded, it will be trained. A decoded tool is retrained from the edges measured on the original training image at the original training time – decoded tools do not retrain from the original training image.

Running The Line Scan Distortion Correction Tool

To run a trained Line Scan Distortion Correction tool, call the tool's **run()** function supplying a run-time image from the line scan camera used to generate the training image as the input image. The supplied image must be the same width as the training image, but it may be of any height.

Note:
If a subregion was used to train the tool, then a subregion with the same width and x-offset must be applied to the run-time images.

The line Scan Distortion Correction tool is intended to be run immediately after image acquisition and before any image analyses are performed. This is the reason why the tool simply copies the source image's *clientFromImage* transform to the destination image. In other words, this is an image coordinate-based tool, not a client coordinate-based tool; the tool ignores client coordinates for the purpose of resampling the pels. Also, this is the reason why the tool does not provide any ways to recover the original image pixel locations from the undistorted (resampled) image pixel locations.

Caliper Tool

This chapter describes the Caliper tool, a vision tool that offers extremely rapid and precise pattern detection and location within a well-defined area of an image. This chapter contains the following sections.

This section and *Some Useful Definitions on page 189* give an overview of the chapter and define some terms you will encounter as you read.

Caliper Tool Overview on page 189 provides an overview of the operations that the Caliper tool performs and defines the concepts and principles that underlie the Caliper tool.

How the Caliper Tool Works on page 193 provides a description of how the Caliper Tool works.

Using the Caliper Tool on page 206 describes how to use the Caliper tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

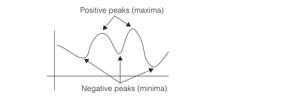
correlation mode: Feature detection based on finding locations of high correlation between a run-time image and a reference image

edge mode: Feature detection based on locating edges or edge peaks within the run-time image.

edge model: Template for the edge pattern that you are searching for with the Caliper tool.

edge pattern candidate: Edge pattern within the run-time image that is being evaluated for its similarity to the edge pattern described by the edge model.

peak: Maximum or minimum in the value of a function or a plot of values; a maximum is a positive peak; a minimum is a negative peak.



projection: Reduction of a two-dimensional array of pixels to a one-dimensional array of pixels. The tool forms a projection by summing the pixel values along a ray within the projection region.

ray: Line drawn in the direction of a projection along which pixel values are summed.

reference image: Image of the feature of interest.

score: Measure of the similarity between an edge pattern candidate found by the Caliper tool and an edge model.

scoring function: Look-up table or function that maps input values to output values within a fixed range.

scoring method: Procedure for evaluating the degree of similarity between an edge pattern candidate and an edge model.

Caliper Tool Overview

The Caliper tool is a vision tool for measuring the width of objects, the location of edges or features, and the location and spacing of pairs of edges in images. The Caliper tool differs from other vision tools in that it requires that you know the approximate location and characteristics of the feature or edges you want to measure or locate.

The Caliper tool is typically used to develop detailed information about the precise locations of features within an object. It is not appropriate for developing information about the shape of objects or features.

The Caliper tool supports the following two methods of locating edges or edge pairs in an image:

- Edge mode, in which edge peaks are detected using a simple 1-D kernel
- Correlation mode, in which correlation peaks between a 1-D sample image and a 1-D reference image are located

Once you have determined the location and orientation of the edge or edge pair of interest within the image, you perform the following steps to apply the Caliper tool:

- Specify a projection region that encloses the edges or edge pairs of interest.
- For edge mode
  - Select a filter size and contrast threshold that isolate the edges of interest.
  - Define an edge model that describes the edge or edge pair of interest.
- For correlation mode
  - Supply a one-dimensional reference image.
- Define the scoring methods that are used to score edge pattern candidates within the image.
- Apply the Caliper tool and interpret the results.

Projections

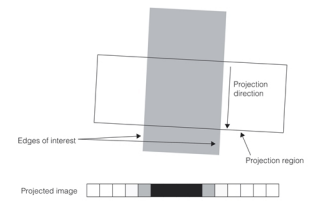
The first step in using the Caliper tool is to specify a projection region within the image to which you want to apply the tool. The Caliper tool depends on a carefully defined *projection region* to isolate just the edge information from a small section of the image.

You specify a projection region for the Caliper tool in either of the following ways:

- Supply an affine rectangle and affine sampling parameters
- Define the location, size, and angles of skew and rotation for the projection region

Typically, the projection region contains the portion of the image with the features of interest. The projection operation sums all the information in the projection region, accentuating the strength of edges that lie parallel to the projection rays and reducing the effects of noise.

The figure below shows how a projection can accentuate the edge information in a two-dimensional image. Notice how the strength of the edge is stronger in the projected image than in the initial image.



Accentuating edge information with projection

For more information on projection, and for examples of how to specify projection regions to accentuate edges of interest, see the chapter *Image Transformation Tools* on page 109.

Once the Caliper tool has produced a projected image, it locates edges using either edge mode or correlation mode.

Edge Mode

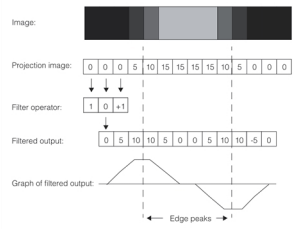
This section describes how the Caliper tool detects edges in edge mode.

Filtering

The Caliper tool uses the projection region to produce a one-dimensional representation of the portion of the image that contains the edges of interest. This one-dimensional projection image will contain not only the edges of interest, but also other edges caused by noise and unwanted information in the original image. Applying a filter to the one-dimensional projection image increases the strength of the edges of interest while at the same time decreasing image noise.

The Caliper tool performs filtering by convolving the one-dimensional projection image with a filter operator. A filter operator is a block of arithmetic operators that is applied sequentially to blocks of pixels within the one-dimensional projection image to produce the pixel values in the filtered image.

The figure below illustrates an example of a simple 3-element filter operator. In this case, the filter operator subtracts the value of each pixel's left neighbor and adds the value of each pixel's right neighbor to produce the pixel value in the filtered image.

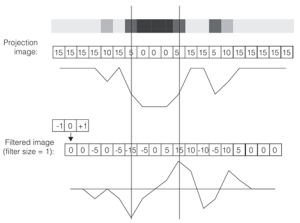


Filtering the projection image

The filtered image no longer visually resembles the input image. However it does have an important new characteristic: a graph of the filtered pixel values reveals that the peaks in the values, both positive and negative, correspond to the location of the edges within the initial image. The Caliper tool uses these peaks in the filtered image to determine the location of edges in the original image.

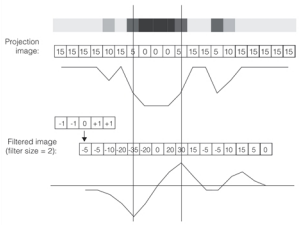
In addition to producing an image with peaks that correspond to edges in the input image, image filtering also removes noise and spurious edges from the input image. The figure below illustrates an example where the image contains two true edges along with spurious edges caused by variations in pixel values around the edge.

When a filter with a size of 1 is applied to the image, both the edges of interest and spurious edges appear as peaks in the filtered projection image.



Filtering with filter size of 1

The figure below shows the effect of applying a filter with a width of 2. The peaks that correspond to the edges of interest are broader, and most of the spurious peaks from the figure above are no longer present in the filtered projection image.



Filtering with filter size of 2

Edge Mode Scoring

Once it has located the edges in the original image, the Caliper tool computes a *score* for each potential edge pattern in the image. The score for each edge pattern in the image is computed based on the similarity between the edge pattern in the image and an ideal edge pattern that you define. The edge patterns in the image are called *edge pattern candidates* while the ideal edge pattern is called the *edge model*. You can define edge models that contain a single edge or a pair of edges.

You control the method that the Caliper tool uses to score edge pattern candidates by performing the following steps:

- Define the edge model that describes the edge or edge pair for which you are searching.
- Define one or more scoring methods that define how edge pattern candidates are to be evaluated for similarity to the edge model.

An edge model is a description of the edge pattern that you expect to see in the image. An edge model includes edge spacing, position, and polarity (dark to light or light to dark).

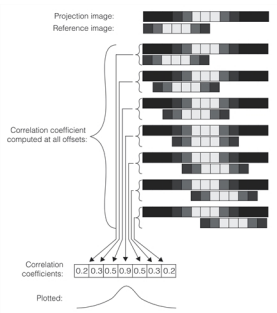
For each edge pattern candidate in the image, the Caliper tool computes a score. You can specify exactly how to evaluate each edge attribute, such as position, strength, or polarity, when computing the overall score for that edge. The tool computes a separate value for each attribute for each edge pattern candidate based on the scoring method that you supply, then computes an overall score for each edge pattern candidate by combining the individual scores.

Correlation Mode

Correlation mode detects features by computing the correlation coefficient between the projection image and a reference image that you supply. The reference image should be an image of the pattern of interest. For example, if you wanted to search for a light-colored lead on a dark background, you would use a reference image similar to that shown in the figure below.

Reference image

At run time, the projection image is constructed as described in the section *Projections* on page 190, then the correlation coefficient is computed between the reference image and the projection image at all possible offsets of the reference image within the projection image. The resulting correlation coefficients are stored in a one-dimensional image. Peaks in this image represent locations of high correlation between the reference image and the projection image. The figure below shows how correlation mode works.



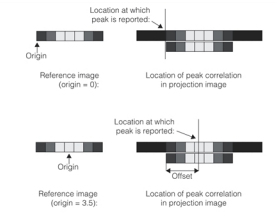
Correlation mode pattern location

Note that the peak in the image shown in the figure above corresponds to the location in the run-time image that most closely resembles the reference image.

Reference Image Origin

By default, the location of the image peak is reported to be the location of the beginning of the reference image within the projection image. You can specify an *origin*, in which case the peak location is offset by the value you specify. The figure

below shows how an origin works.



Specifying an origin

Note: You actually specify the origin as a 2-dimensional location within the reference image.

Acceptance and Peak Separation Thresholds

The Caliper tool lets you specify two thresholds in correlation mode.

The *acceptance threshold* is the minimum correlation value for a correlation peak. Only peaks with correlation coefficient values greater than the threshold you specify are scored by the Caliper tool.

The *peak separation threshold* is the minimum distance between correlation peaks. The Caliper tool only scores peaks that are separated from one another by the peak separation threshold you specify.

Correlation Mode Scoring

Correlation mode scoring performs the same function as edge mode scoring; it evaluates correlation peaks within the image to determine the best peak. Because correlation mode does not use edges, there is no edge model or edge pattern candidates. Instead, scoring is based on the height and location of correlation peaks.

Caliper Tool Results

The Caliper tool returns the overall score and position of all edge pattern candidates within the image that receive scores above a minimum acceptance score that you supply.

You can also request intermediate results from the Caliper tool such as the projection image, the filtered image, and a list of all individual edge peaks in the image.

How the Caliper Tool Works

This section describes how the Caliper tool works.

Specifying the Projection Region

When specifying your projection, you define a rectangular or parallelogram-shaped region within the input image. You should define this rectangular region so that

- The edges of interest in the image are parallel to the projection direction.
- The projection region encloses as much of the edge or edges as possible while enclosing as little additional area as possible.

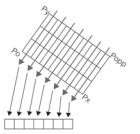
You define a projection region in either of the following ways:

- By an affine rectangle and an affine rectangle sampling parameters
- By specifying the location, width, height, and angles of skew and rotation

Specifying an Affine Rectangle and Sampling Parameters

You can specify the projection region as an affine rectangle and affine rectangle sampling parameters. The projection direction is defined as being parallel to the Po-Py side of the affine rectangle, and the number of Po-Px divisions specified by the affine sampling parameters defines the number of samples in the projection region.

The figure below shows how an affine rectangle and affine rectangle sampling parameters determine the projection region.

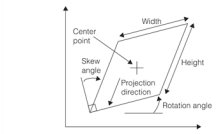


Project region defined by affine rectangle and sampling parameters

For information on affine rectangles and affine sampling parameters, see the chapter on *Image Transformation Tools* on page 109.

Specifying a Projection Region Directly

You can specify the projection region directly by supplying the location of its center point, its width, its height, its angle of rotation, and its angle of skew. The figure below shows how you define a projection region using these parameters.



Specifying a projection region

For more information on specifying a projection region directly, see the chapter *Projections* on page 190.

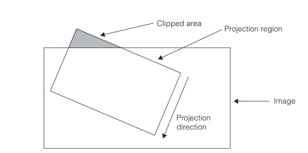
Sampling Method

If you provide an affine rectangle and affine rectangle sampling parameters to define the projection region, the sampling type that the Caliper tool uses is determined by the sampling type within the affine rectangle sampling parameters.

If you specify the projection region directly, then you must specify either uninterpolated or interpolated sampling. For information on specifying a sampling method, see the chapter *Image Transformation Tools* on page 109.

Clipped Projections

When projections are rotated or skewed, the projection region can extend outside the boundary of the image, particularly if the projection region is close in size to the image. The figure below shows a projection region that extends outside the image; the clipped area is shown in grey.



Clipped projection

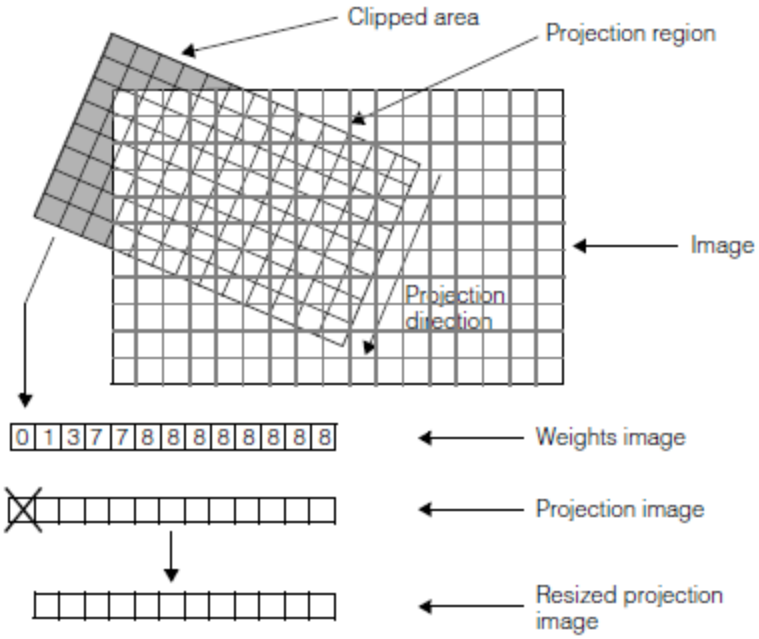
You can choose how the Caliper tool handles a clipped projection region from among the following methods:

- Throw an error if the projection image clips
- Automatically reduce the size of the projection image
- Return a weights image

If you do not specify a clipping method, the Caliper tool will automatically reduce the size of the projection image when clipping occurs.

Auto-Clipping Mode

You can configure the Caliper tool to automatically adjust the projection region if it detects clipping. The Caliper tool detects clipping by computing a *weights image* when it computes the projection image. The weights image is a one-dimensional image in which each pixel is set to be the number of pixels from the input image which contributed to the corresponding pixel in the projection image. The figure below shows how the weights image is computed.



Automatic clipping based on a weights image

If you specify auto-clipping mode, the Caliper tool resizes the projection image so that no pixels with weights of zero are included.

The Caliper tool returns a status flag that indicates whether or not clipping occurred.

When the Caliper tool produces the filtered image and performs peak detection, it automatically normalizes the projection image based on the number of source image pixels that contributed to each projection image pixel.

Returning a Weights Image

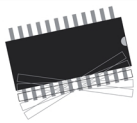
You can also obtain the weights image produced by the Caliper tool and perform your own processing of the image based on the weights image. The weights image is computed as show in the figure in the Auto-Clipping Mode section above.

Scan Mode

You can configure the Caliper tool to automatically generate and score projection images across a range of angles. This lets you avoid having to precisely compute the angle at which the image lies.

Scan mode works by applying the caliper at a specified number of incremental angles within the range that you specify and returning the results that obtained the highest average score for all of the edges or edge pairs scored

The figure below shows an example of scan mode. The projection region which produces the highest average score for all edges or edge pairs is used.



Scan mode

You can control the number of intermediate angles at which the Caliper tool is applied in scan mode.

Supplying a Projection Image as Input

If you use another vision tool, such as the Labeled Projection tool, that produces a one-dimensional image as output, you can supply that one-dimensional image as input to the Caliper tool. When you supply such an image, the Caliper tool applies whatever filtering and scoring parameters you specify.

You can use this feature to locate edges in projection images that are not based on standard projection regions.

For more information on the Labeled Projection tool, see the chapter *Labeled Projection Tool* on page 80.

Edge Mode Edge Detection

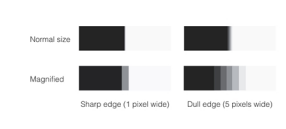
Edge mode edge detection uses a filter operator to extract edge peaks from the projection image.

Specifying the Filter

The filter size and type that you specify for the Caliper tool controls how the tool removes noise from the input image and how it accentuates the peaks of interest in the image.

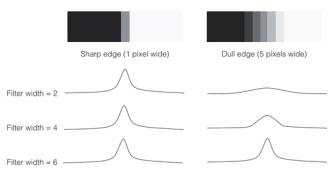
Filter Size

You should specify a filter size that closely matches the size of the edges in your input images. The size of an edge is the number of pixels wide that the edge is. Edges can be *sharp*, in which case the edge only spans one or two pixels, or edges can be *dull*, in which case they might span many pixels. The figure below illustrates examples of sharp and dull edges; the figure illustrates both types of edges at different magnifications.



Sharp and dull edges

When you specify a filter size that is close to or greater than the edge size, the Caliper tool produces stronger edge peaks in the filter image. If you specify a filter size that is too small, the filter image will contain broad, low peaks. The figure below illustrates the effect of applying filters of different sizes to sharp and dull edges.



The effect of filter size and edge size on peak size

The figure above illustrates that if you select a filter size that matches or that is greater than the size of the edges of interest within the image, the desired peaks in the filtered image will be sharper and there will be fewer spurious peaks.

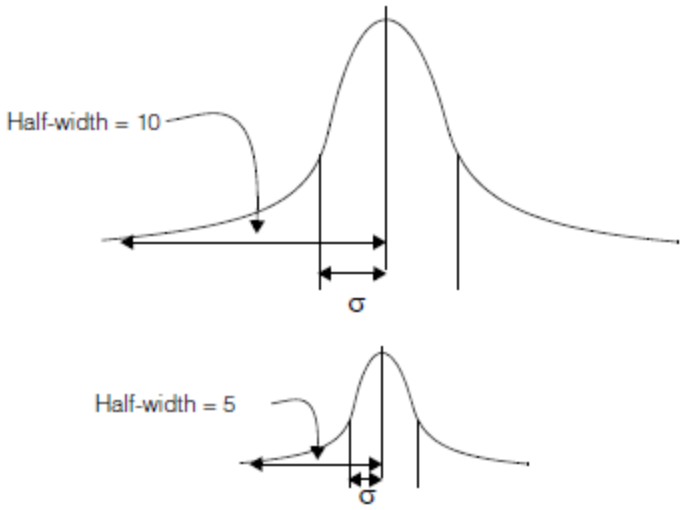
If you have multiple edges within the projection region and you select a filter size that is large enough to include a neighboring edge, the included pixel values of the neighboring edge will cause the degradation of the current edge's peak in the filter image. Therefore, make sure you specify a filter size that does not include neighboring edges that are

expected to appear in the input image. In other words, the upper limit of the filter size is determined by the minimum expected distance between edges in the input image.

Filter Type

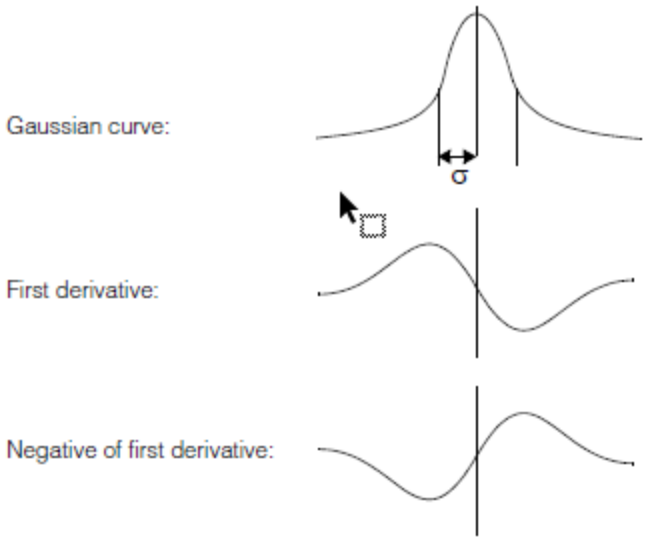
The Caliper tool supports a filter kernel that approximates the negative of the first derivative of a Gaussian distribution. This kernel provides better smoothing than traditional boxcar filters.

You specify the *half-width* of the filter, where the half-width is equal to four times the sigma of the Gaussian distribution. The figure below illustrates how you specify the width of the filter.



Filter size for a Gaussian filter

The filter itself is constructed by taking the negative of the first derivative of the Gaussian curve, as shown in the figure below.



Constructing the filter

Peak Detection

The Caliper tool lets you define a minimum peak height or *contrast threshold*. Any peaks that are lower than your minimum peak height are excluded from the results. This allows you to limit your analysis of edges in the image to just those edges of a certain magnitude.

You specify the contrast threshold as the difference in normalized pixel values between the two sides of the edge.

Correlation Mode Pattern Detection

Correlation mode edge detection computes the correlation or similarity between a reference image that you supply and the projection image produced by applying the projection parameters to the run-time image.

Reference Images

To use correlation mode, you must supply a reference image of the one-dimensional pattern you are looking for. You can supply a reference image using any of the following methods:

- Supply a one-dimensional image to use as the reference image.
- Supply a two-dimensional image and a set of affine rectangle sampling parameters
- Supply a run-length encoded image with optional smoothing parameters.
- Use a pre-defined reference image.

Each of these methods is described in the following sections.

One-Dimensional Image

You can supply the reference image as a normal image. You can construct this image synthetically, or you can create it as a projection image from an image you acquire, then manipulate using the Projection tool.

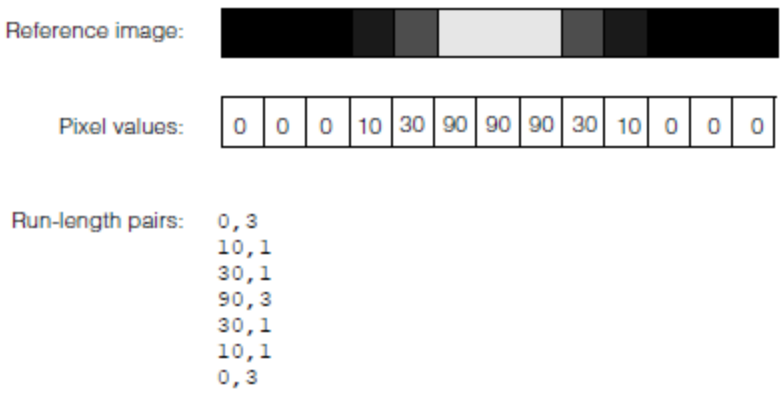
Two-Dimensional Image and Sampling Parameters

You can supply a standard two-dimensional image together with affine rectangle sampling parameters. The Caliper tool will automatically construct the reference image using the image and parameters you supply.

For more information on affine rectangle sampling parameters, see the chapter *Image Transformation Tools* on page 109.

Run-Length Encoded Image

You can supply the reference image in the form of a run-length encoded image. In this case, you supply a list of integer pairs. The first element of each pair is a pixel value. The second element of each pair is the run-length. shows how you use a run-length encoded image.



Run-length encoded reference image

In addition to the list of run-length pairs, you can also specify a value for the size of a Gaussian kernel which will be used to smooth the data you supply. You can use this to reduce noise in a reference image.

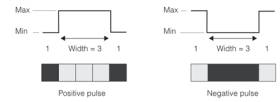
Pre-Defined Reference Images

You can use one of four pre-defined reference images by specifying the reference image type and dimensions. The predefined reference images are appropriate for simple edge pairs.

You can specify any of the following types of predefined reference images:

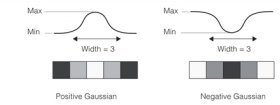
- Positive pulse
- Negative pulse
- Positive Gaussian
- Negative Gaussian

The positive and negative pulse images are simple boxcar images. You specify the low and high values and the width of the pulse, as shown in the figure below. The overall reference image is two pixels larger than the pulse width you specify.



Positive and negative pulse reference images

The positive and negative Gaussian images constructed by applying the Gaussian Sampling tool to a positive or negative pulse image. You specify the low and high values and the width of the pulse, as well as the smoothing value to use with the Gaussian Sampling tool. As with the pulse images, the overall width of the resulting image is always equal to the pulse width+2, as shown in the figure below.



Positive and negative Gaussian reference images

You can also specify a default smoothing value for the Gaussian reference image, in which case the tool computes a smoothing value such that the resulting Gaussian curve has a sigma equal to one fourth of the pulse width.

Normalized versus Relative Correlation Coefficients

In addition to specifying a reference image, you must also specify the type of correlation coefficient to compute. The Caliper tool supports two types of correlation coefficients:

- Normalized correlation
- Relative correlation

Each type of correlation coefficient is described in the following sections.

Normalized Correlation

Normalized correlation is a measure of the geometric similarity between an image and a model, independent of any linear differences in image or model brightness. The normalized correlation value does not change in either of the following situations:

- If all image or model pixels are multiplied by a constant
- If a constant is added to all image or model pixels

Mathematically, the correlation coefficient r of a model and a corresponding portion of an image at an offset (u) is given by

$$r\left(u\right) = \frac{\left[N \sum_i I_i M_i - \left(\sum_i I_i\right)\left(\sum_i M_i\right)\right]}{\sqrt{\left[N \sum_i I_i^2 - \left(\sum_i I_i\right)^2\right]\left[N \sum_i M_i^2 - \left(\sum_i M_i\right)^2\right]}}$$

where:

N is the total number of pixels

I_i is the image pixel at $(u+x_i)$

M_i is the corresponding model pixel at the relative offset (x_i)

The value of r is always in the range -1 to 1, inclusive. A value of 1 signifies a *perfect match* between the image and model. Specifically, if $r = 1$, there exist some values a and b such that for all i: $I_i = aM_i + b, a > 0$

A value of -1 signifies a “perfect mismatch,” which is the same as a perfect match except that a < 0: the feature found is the *negative* of the model. The negative of a model or image is a corresponding image in which the sense of light and dark has been reversed.

Relative Correlation

Relative correlation is independent of linear *additive* and *multiplicative* differences in model brightness and *additive* differences in image brightness. If all image pixels are *multiplied* by a constant, however, the relative correlation value is also multiplied by that constant. That is, the correlation coefficient varies linearly with the contrast of the feature.

Mathematically, the relative correlation coefficient c of a model and a corresponding portion of an image at an offset (u) is given by $c\left(u\right) = \frac{[N \sum_i I_i M_i - (\sum_i I_i)(\sum_i M_i)]}{N\sqrt{N \sum_i M_i^2 - (\sum_i M_i)^2}}$

where:

N is the total number of pixels

I_i is the image pixel at $(u + x_i)$

M_i is the corresponding model pixel at (x_i)

As defined by this equation, the relative correlation value has the same size as the grey values in the image. That is, for image pixels having 8 bits (0 - 255), the correlation value will have 8 bits signed (-128 to +127). This does not mean that 127 is a perfect match; there is no such thing as a perfect match with relative correlation. The value 127 is the theoretical maximum value the formula could obtain for certain models if the image were sufficiently bright.

Edge Mode Scoring

Once the Caliper tool has filtered the projection image and produced a list of the edge peaks that exceed the contrast threshold that you specify, the tool computes a *score* for each edge pattern candidate within the image. This score lets your application determine which of the edge pattern candidates in the image represent instances of the actual edge pattern of interest, as defined by your edge model.

The Caliper tool computes the score for each edge pattern candidate by comparing the edge pattern candidate with the edge model, based upon a set of scoring criteria that you supply. These criteria are called *scoring methods*.

A scoring method has two parts:

- A *scoring method type* that defines what measurement of the edge you want to evaluate
- A *scoring function* that defines the relationship between the raw measure and the mapped score that will be generated for the scoring method

You can define several scoring methods. The Caliper tool applies all the scoring methods to each edge pattern candidate within the image and returns an overall score for each edge pattern candidate. By defining appropriate scoring methods, you can ensure that the edge pattern candidate with the highest score will be the edge pattern of interest.

Edge Model

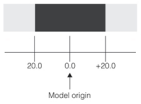
To evaluate whether an edge pattern candidate in the image is a good match for the edge pattern that you are seeking, you must define a model that describes the edge pattern of interest. You define an edge model by specifying the polarity (light-to-dark or dark-to-light) and position of each edge in the model. You specify the positions of edges relative to a *model origin*. The Caliper tool lets you define edge models with a single edge or a pair of edges.

The table below describes an edge model with a symmetric pair of edges. The origin of the model is centered between the two edges.

Edge	Position	Polarity
1	–20	Light to dark
2	+20	Dark to light

Edge model

The figure below shows an idealized graphical representation of the edge model described in the table above. Note that the position of the model origin is implied by the values of the edge positions.



Idealized representation of an edge model

Scoring Method Types

The Caliper tool bases the score of an edge pattern candidate on how different the edge pattern candidate is from the edge model. You can specify the particular measure that the tool uses to assess this difference. The types of measures that you can specify are called *scoring method types*. The available scoring method types are

- Position
- Size
- Contrast
- Straddle

When the tool applies a scoring method type to an edge pattern candidate the result is a raw score. The raw score is different for different scoring method types. The scoring method types and their raw scores are described in the following sections.

Positional Scoring Methods

You can score an edge pattern candidate based on the position of the edge pattern candidate relative to the application point of the projection region you specified for this Caliper tool. The position is defined as the distance between the application point and the model origin point within the edge pattern candidate.

Note:
In most cases, the application point is the point within the projection region that corresponds to the center of the affine rectangle used to specify the projection region.

If you supply a one-dimensional image as input to the Caliper tool, then you must also supply the point within that image to use as the application point.

If you expect the edge of interest to be a specific distance from the application point, then you can define an absolute positional scoring method and the raw score will be expressed as an absolute distance in pixels.

If you are using an edge pair model and you would like to consider the variation in position between the edge pattern candidate and the application point relative to the size of the model, you can define a relative positional scoring method.

In this case the raw score will be normalized so that a value of 1.0 means that the distance was equal to the size of the model.

Size Scoring Methods

If you are using an edge pair model, you can score edge pattern candidates based on how much the width between the edges of the edge pattern candidate varies from the width between the edges of the edge model. You can define a size scoring method to be absolute, in which case the raw score will be returned as an absolute size difference in pixels.

If you would like to consider the size difference relative to the size of the model, you can define a relative size scoring method. In this case the raw score will be normalized so that a value of 1.0 means that the size difference was equal to the size of the model.

Contrast Scoring Methods

You can score edge pattern candidates based on the contrast of the edge pattern candidates. The contrast of an edge is expressed in terms of the change in pixel values divided by the size of the edge in pixels. The raw score for a contrast scoring method is normalized so that a value of 1.0 is equal to a contrast of 256 (the maximum possible value for contrast). If you specify an edge pair model, the raw score is the average of the contrast for the two edges.

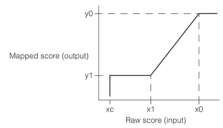
Straddle Scoring Methods

If you are using an edge pair model, you can score edge pair candidates based on whether or not the two edges lie on either side of the application point. This type of scoring method can be used to find objects defined by a pair of edges that are expected to lie under the application point. The raw score is returned as 1.0 if the application point is straddled by the edges, 0.0 if the application point is not straddled by the edges.

Scoring Functions

For each scoring method, the selected scoring method type produces a raw score. You control the effect that this raw score has on the overall score for an edge pattern candidate by defining a *scoring function*. A scoring function maps a raw score to a mapped score. The mapped scores for each scoring method for an edge pattern candidate are combined to form the overall score for that edge pattern candidate.

You define a scoring function by specifying low and high input and output values; the Caliper tool creates a scoring function based on the values you supply. The figure below shows a scoring function.



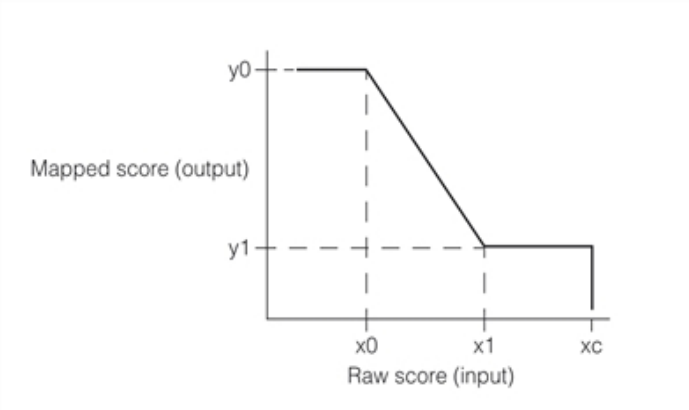
Scoring function

You define the scoring function by defining values for *xc*, *x1*, *x0*, *y1*, and *y0*. The Caliper tool uses the scoring function you define to map raw scores to mapped scores as follows. Raw scores above *x0* are mapped to a score of *y0*. Raw scores below *xc* are mapped to a score of 0. Raw scores between *xc* and *x1* are mapped to a score of *y1*. Raw scores between *x1* and *x0* are mapped linearly to the range of scores between *y1* and *y0*.

The values you define for *y0* and *y1* must be between 0.0 and 1.0. You can specify any values for *xc*, *x1*, and *x0*, including negative values. You would specify negative values for one or more of the *xc*, *x1*, and *x0* points if you expected raw scores less than zero.

The scoring function illustrated in the figure above would be appropriate for a case where higher raw scores should produce higher mapped scores, such as a contrast scoring method where greater the edge contrast should produce a higher score.

In other cases, such as a positional scoring method where the edge pattern candidate is expected to be at or near the application point, a higher raw score should result in a lower mapped score. In a case such as this, you specify a value for *x1* that is greater than *x0*, and a value for *xc* that is greater than *x0*. The figure below shows a scoring function that is appropriate for a case where higher raw scores should result in lower mapped scores.



Scoring function with x1 greater than x0

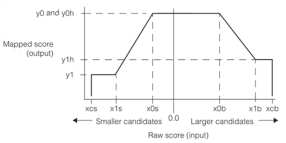
You must specify one and only one scoring function per scoring method, although you can specify any number of geometric scoring methods per Caliper tool.

Defining a Two-Sided Scoring Function

If you are using an edge pair model and you specify a two-sided size scoring method type, you can define a scoring function that scores edge pattern candidates that are smaller than the edge model differently from edge pattern candidates that are larger than the edge model. This type of scoring function is called a *two-sided scoring function*.

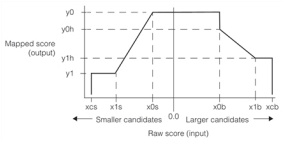
The figure below shows an example of a two-sided scoring function that is more tolerant of edge pattern candidates that are larger than the edge model than of edge pattern candidates that are smaller. If the edge pattern candidate is exactly the same size as the edge model, the raw score will be 0.0; if the edge pattern candidate is smaller than the edge model, the raw score will be less than 0; if the edge pattern candidate is larger than the edge model, the raw score will be greater than 0.

If you specify a two-sided scoring function, the raw score is normalized so that an edge pattern candidate that is smaller than the edge model by an amount equal to the size of the edge model receives a raw score of –1.0 and an edge pattern candidate that is larger than the edge model by an amount equal to the size of the edge model receives a raw score of 1.0.



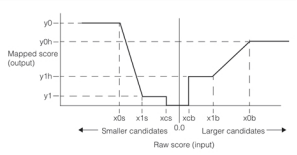
Two-sided scoring function

The figure above shows a two-sided scoring function with distinct values for *y1* and *y1h* but the same values for *y0* and *y0h*. While it is possible to specify different values for *y0* and *y0h*, the resulting scoring function contains a discontinuity at *x0b*. The figure below shows a scoring function where *y0h* is different than *y0*.



Two-sided scoring function

You can also specify a two sided scoring function for cases where raw score values *outside* of a range of values receive higher scores. The figure below shows an example of this type of two-sided scoring function.



Two-sided scoring function with low output values in the center

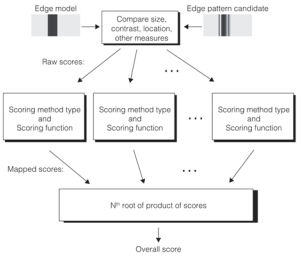
Computing the Overall Score for an Edge Pattern Candidate

For each edge pattern candidate, once the Caliper tool has computed the raw score for each scoring method and applied the scoring function for that scoring method it computes an overall score for the edge pattern candidate.

The Caliper tool computes the overall score for an edge pattern candidate by taking the N^{th} root of the product of the mapped scores. For example, if you defined four scoring methods, the Caliper tool would multiply the four scores and take the 4^{th} root of the result.

Note that if any single scoring function produces a value of 0, then the overall score for the edge pattern candidate will also be 0.

The figure below shows the overall process of computing the score for an edge pattern candidate.



Scoring an edge pattern candidate

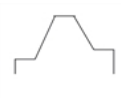
Correlation Mode Scoring

Correlation mode is essentially the same as edge mode scoring except that you cannot use a scoring method that is designed for edge pairs or edge contrast (since there are no edges in the output produced by correlation mode).

Scoring Methods

The Caliper tool includes several pre-defined scoring methods. Each of these scoring methods is described in the table below.

Scoring Method Type	Raw Score (Input Measure)	Scoring Function
Normalized size difference	For edge pairs only: the absolute difference between the width of the edge model and the width of the edge pattern candidate, divided by the width of the edge model. This is defined as: $\frac{\lvert w-d \rvert}{w}$ where d is the width of the candidate, and w is the width of the model.	One-sided: 
Normalized size	For edge pairs only: the width of the edge pattern candidate divided by the width of the edge model. This is defined as: $\frac{d}{w}$ where d is the width of the candidate, and w is the width of the model.	One-sided: 

Scoring Method Type	Raw Score (Input Measure)	Scoring Function
Normalized size difference (two-sided)	For edge pairs only: the absolute difference between the width of the edge model and the width of the edge pattern candidate, divided by the width of the edge model. This is defined as: $\frac{(w-d)}{w}$ where d is the width of the candidate, and w is the width of the model.	Two-sided: 
Normalized position	For edge pairs only: the distance between the origin of the edge pattern candidate and the application point, divided by the width of the edge model. This is defined as $\frac{\lvert a\rvert}{w}$ where a is the application point, measured with respect to the origin of the candidate, and w is the width of the model.	One-sided: 
Position	The distance between the origin of the edge pattern candidate and the application point. This is defined as $\lvert a\rvert$ where a is the application point, measured with respect to the origin of the candidate.	One-sided: 
Straddle	For edge pairs only: whether or not the edge model straddles the application point If the application point lies between the two edges, the value is 1.0. If it does not, the value is 0.0.	None (mapped score is same as raw score)
Contrast	For a single edge model: the absolute edge contrast; for an edge pair model: the average absolute contrast of the two edges Defined as: $\frac{\Delta}{w}$ where Δ is the absolute change in pixel value across the edge and w is the width of the edge; the raw score is in the range of 0 through 255.	One-sided: 

Pre-defined scoring methods

Evaluating Results

For each edge pattern candidate in the filtered projection image that exceeds the contrast threshold you supply, up to the number of results that you specify, the Caliper tool returns the following information:

- The position of the edge pattern candidate relative to the projection region application point
- The overall score for this edge pattern candidate, computed using the scoring methods you supplied
- The polarity and contrast of all edges in the edge pattern candidate
- The individual scores for each of the scoring methods you supplied

In addition, you can request the following additional information from the Caliper tool:

- The polarity and contrast of each individual edge in the filtered projection image
- The unfiltered projection image itself
- The filtered projection image itself
- The number of edge pattern candidates that exceeded the contrast threshold

You can use this optional information to perform your own processing and analysis of the intermediate data generated by the Caliper tool.

Using the Projection Image Transformation

The Projection image produced by the Caliper tool has a transformation object associated with it. You can use this transformation to map points within the projection region to points within the input image.

For more information on using the projection image transformation, see the chapter *Image Transformation Tools* on page 109.

Using Nonlinear Client Coordinates

If you supply it with a run-time image that has a nonlinear client coordinate system, the Caliper tool computes a local linearization of the nonlinear client coordinates across the projection region that you specify to the tool. The Caliper tool computes this linearization by performing a least-squares fit of five points within the region.

The resulting linear transformation is used to construct the affine rectangle used for sampling.

The Caliper tool detects edge peaks in the linearized and transformed image produced as described above. Before the tool performs any scoring, however, it converts each edge and peak location back into the nonlinear client coordinate system. All Caliper results are reported in client coordinates.

For more information, see the chapter *Using CVL Vision Tools* in the *CVL User's Guide*.

Using the Caliper Tool

This section describes the overall procedure for using the Caliper tool.

In general, you will follow these steps to use the Caliper too in edge mode:

- Define a projection region that encloses the edges of interest.
- Define a filter width that accentuates the edges of interest.
- (Optional) Define a contrast threshold that excludes spurious edges.
- Define one or more scoring methods.
- Apply the tool to an input image.
- Analyze the results.

In general, you will follow these steps to use the Caliper too in correlation mode:

- Define a projection region that encloses the edges of interest.
- (Optional) Define an acceptance threshold to specify the minimum peak height.
- Define a single contrast scoring method.
- (Optional) Define an acceptance threshold to specify the minimum peak height.
- Apply the tool to an input image.
- Analyze the results.

Edge Tool

This chapter describes the Edge tool, a tool that can isolate and enhance edge information in an image.

This section and *Some Useful Definitions on page 207* give an overview of the chapter and define some terms you will encounter as you read.

Edge Tool Overview on page 207 describes the basic capabilities of the Edge tool.

How the Edge Tool Works on page 210 provides information that helps you make the most effective use of the Edge tool.

Using the Edge Tool on page 212 outlines the steps to using the Edge tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

edge: Boundary between two regions of different pixel values

edge angle: Orientation of an edge at an edge pixel, expressed as the angle between the client coordinate system x-axis and a line drawn perpendicular to the edge going from dark to light

edge angle image: Image created by replacing each pixel in an input image with a pixel whose value is equal to the edge angle of the pixel in the input image

edge magnitude: Strength of an edge at an edge pixel, expressed as the difference between pixel values on the two sides of the edge

edge magnitude image: Image created by replacing each pixel in an input image with a pixel whose value is equal to the edge magnitude of the pixel in the input image

edge pixel: Pixel through which an edge passes

edgelet: Data structure describing the sub-pixel location of an edge along with its angle and magnitude

featurelet: A common data structure used by several vision tools that describes the sub-pixel location of an edge along with its angle, weight and magnitude. Edge tool results can be converted into featurelet chain sets.

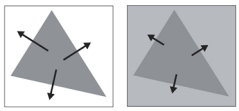
Edge Tool Overview

The Edge tool locates edges in an image. It removes constant and slowly varying backgrounds from an image, leaving edges as the only image feature, and it calculates the magnitudes and angles of edges.

Edges in Images

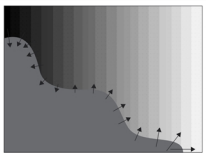
The edge of an object is marked by a change in grey level from dark to light or from light to dark, and it is characterized by two properties: *magnitude* and *angle*. The magnitude of an edge is the amount of change in grey level when crossing the edge. The angle of an edge is the angle of a line drawn perpendicular to the edge.

The figure below shows two triangles, each with arrows whose directions indicate the edge angles and whose lengths indicate the edge magnitudes. Each triangle has the same edge angles, but the edge magnitudes of the triangle on the left are higher than those of the triangle on the right due to the different background grey level.



Edge orientation and magnitude

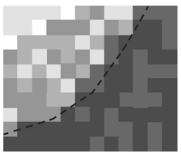
The angle and magnitude of an edge can change continuously. The figure below shows a curved edge. In addition to a varying angle, it has a changing edge magnitude, denoted by the arrow length, due to shifts in the background grey level.



Edge orientation and magnitude varies continuously

Edge Pixels

An edge pixel is a pixel whose grey level is different from that of an adjacent pixel. The figure below shows an array of pixels with varying grey levels. An image edge, denoted by a dashed line, runs diagonally across the figure from bottom-left to top-right. However, almost every pixel in the figure can be considered an edge pixel, as the majority of pixels have neighbors with different grey levels.



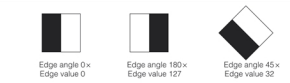
Edge passes through pixels (magnified)

Edge Magnitude Image

An edge magnitude image is an image in which the grey level of each pixel corresponds to the pixel edge magnitude in the input image. In general, a large edge magnitude will generate a lighter grey level, and a small edge magnitude will generate a darker grey level.

Edge Angle Image

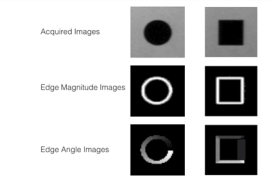
An edge angle image is an image in which the grey level of each pixel corresponds to the pixel edge angle in the input image. Edge angles are expressed relative to the *image coordinate system*. An edge angle of 0° indicates a dark-to-light edge perpendicular to the image coordinate system x-axis. Edge angles increase counter-clockwise. Edge angle values are scaled to the range 0-255. The figure below shows examples of different edge angles and the values used to represent them in an edge angle image.



Edge angle and value

Sample Edge Magnitude and Edge Angle Images

The figure below shows acquired, edge magnitude, and edge angle images for a circle and a square. Notice in the edge angle images that the grey levels are constant along each side of the square but change continuously around the perimeter of the circle.



Edge magnitude and angle images

Peak Detection

Each of the edge magnitude images shown in *Edge magnitude and angle images* on page 209 contain many edge pixels around the location of the edge. The figure below shows the edge magnitude image magnified.

Edge magnitude image (magnified)

Edges in images are often blurred. This smearing can be present in the object itself; it can be due to shadows caused by lighting; or it can result from the true edge of an object falling in the middle of a pixel. Using *peak detection*, the Edge tool enhances images so that the true edge can be isolated from nearby insignificant edge pixels.

Whole-Pixel Peak Detection

The peak detection method used by the Edge tool uses a 3-point angle-sensitive neighborhood operator. An edge pixel is defined to be an *edge peak* if it has a greater edge magnitude than both of the adjacent pixels that lie in the edge direction of the pixel. The figure below shows the different possible peak detection neighborhoods.

Peak detection neighborhoods

When the Edge tool performs whole-pixel peak detection, it creates an output edge magnitude image in which all non-peak pixels are set to 0.

The figure below shows an example of the effect of performing peak-detection on an edge magnitude image and edge angle image.

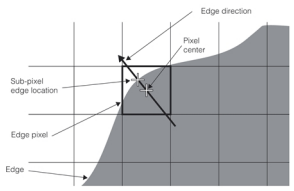
Peak detection

After peak detection, the edge is only one or two pixels wide. In the raw edge magnitude image, the edge is four or five pixels wide. The peak detected edge provides an accurate measure of the true edge in the input image.

Sub-Pixel Edge Peak Detection

In addition to generating whole-pixel edge peak locations, the Edge tool can also determine the locations of edge peaks with sub-pixel accuracy. If you request sub-pixel edge peak locations, the Edge tool returns a list of *edgelets*. An edgelet is simply the sub-pixel location of the true edge peak associated with an edge pixel, along with the edge magnitude and angle.

An edgelet's location is provided with sub-pixel accuracy in both the x- and y-direction, although the only meaningful sub-pixel information about an edge is the one-dimensional offset along a line perpendicular to the edge and passing through the center of the edge pixel, as shown in the figure below.



Sub-pixel edge location

The collection of sub-pixel locations found in an input image are returned as an *edgelet set*, rather than an image. If the programming interface you are using supports transformations, the edgelet set also includes a transformation that lets you map edgelet points to the client coordinate system of the input image.

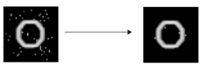
Edge Hysteresis Thresholding

The edge magnitude images produced from most real-world images contain spurious or noise edge pixels. Often these edge pixels are the result of video noise, reflections, or other image imperfections. You can eliminate spurious pixels by applying a simple threshold to the edge magnitude image.

Simple edge thresholding often removes actual edges of interest in addition to spurious edges. Because the actual edges of interest in an image are usually composed of a collection of adjacent edge pixels, you can eliminate spurious pixels without eliminating actual edges by applying *edge hysteresis thresholding* to an edge magnitude image.

Edge hysteresis thresholding eliminates edge pixels that are below a certain magnitude that are not adjacent to other edge pixels above a certain magnitude in the edge magnitude image. This has the effect of preserving the contiguous edge pixels that make up true edges of interest in the image while eliminating edge pixels that have resulted from noise or other image defects.

The figure below shows the effect of edge hysteresis thresholding on an edge magnitude image.



Edge hysteresis thresholding

You can apply edge hysteresis thresholding to any edge magnitude image, but best results will be obtained by using one that has been peak-detected. The figure below shows the effect of edge hysteresis thresholding on a peak-detected edge magnitude image.

Edge hysteresis thresholding on peak-detected image

How the Edge Tool Works

This section describes how the Edge tool works. You can use the information in this section to get the most out of the Edge tool.

Computing a Pixel's Edge Magnitude and Angle

The edge magnitude and edge angle of a given pixel depends on the pixel values of the eight nearest neighbor pixels (upper, lower, left, right, and four diagonal neighbors) using the 3x3 Sobel operator. The figure below shows the 3x3

Sobel operator.

3x3 Sobel operator

The Edge tool calculates the edge angle and magnitude of the center pixel, *E*, using the information from the neighboring pixels using the following steps:

1. The Edge tool calculates the edge magnitudes in the x and y directions (*MagX* and *MagY*) using the formulas:
   $$\mathbf{MagX} = \frac{(C-A)+2\times(F-D)+(J-G)}{4}$$
   $$\mathbf{MagY} = \frac{(G-A)+2\times(H-B)+(J-C)}{4}$$
2. Using the x and y edge magnitudes, the Edge tool calculates the overall edge magnitude, S, using the Pythagorean theorem: $S = \sqrt{\mathbf{MagX}^2 + \mathbf{MagY}^2}$
3. The Edge tool calculates the edge angle, θ, according to the equation $\theta = \mathbf{atan\,2}(\mathbf{MagY}, \mathbf{MagX})$ which yields the edge angle. Because the edge angle is derived directly from the image pixels, edge angles are relative to image coordinates, *not* client coordinates. Also, angles are mapped to the range of values 0-255.

Producing Edge Angle and Magnitude Images

The edge magnitude and edge angle images are constructed by applying the Sobel edge detection operator to an input image, then constructing a pair of images where each pixel is replaced with either the angle or magnitude of the corresponding pixel in the input image.

Ordinarily, applying the Sobel edge detector operator produces an image that is smaller than the input image by two pixels in both the x- and y-direction. However, if the input image is associated with a root image that contains additional pixels outside the boundaries of the supplied image, the Edge tool will use these pixels and will return edge magnitude and edge angle images that are the same size as the input image.

In all cases, the Edge tool sets the client coordinate systems of both the edge angle image and the edge magnitude image such that locations correspond to locations in the input image.

Edge Magnitude Scaling Factor

A strong edge can produce an edge magnitude greater than can be represented in a single pixel. For example, the strongest edge magnitude that can be produced using an 8-bit input image is 286 while the largest value that can be stored in an 8-bit pixel is 255.

You can prevent edge magnitude values from overflowing the pixels in the edge magnitude image by specifying an edge magnitude scaling factor. All edge magnitudes are scaled by the factor you supply before being stored in the magnitude image.

If your edge magnitude image has the same pixel size as the input image, you can guarantee that there will be no overflow by specifying a scale value of 0.89.

For a low-contrast image, you can specify a scale value greater than 1. This expands the range of values in the edge magnitude image, although it can cause magnitude values to overflow.

Note: The values of pixels in the angle image that correspond to pixels in the magnitude image with values less than the edge magnitude scaling factor (rounded to the nearest integer) are undefined.

Peak Detection and Edgelet Lists

The whole-pixel edge detection works exactly as described in the section *Peak Detection* on page 209. The sub-pixel edge detection system uses proprietary Cognex technology to determine sub-pixel edge locations.

Edge Magnitude Threshold

If you use the Edge tool to construct an edgelet set, you can specify that only edge peaks with values above a threshold be included in the edgelet set. This threshold does not affect the values in the edge magnitude image produced by peak detection, only the edgelets in the edgelet set.

Edge Hysteresis Thresholding

Edge hysteresis thresholding lets you eliminate spurious edges without affecting weak edges of interest. To use edge hysteresis thresholding, you specify a low threshold and a high threshold. All pixels with edge magnitudes greater than or equal to the high threshold are included in the magnitude image. All pixels with edge magnitudes below the low threshold are excluded from the magnitude image.

Pixels with edge magnitudes greater than or equal to the low threshold that are 8-connected to another pixel with a magnitude greater than or equal to the high threshold, either directly or through another 8-connected pixel with a magnitude between the low and high thresholds, are included in the magnitude image.

Note: You cannot apply edge hysteresis thresholding to an edgelet list, nor can you construct an edgelet list from a magnitude image created using edge hysteresis thresholding.

Using the Edge Tool

The following sections provide information to help you use the Edge tool.

General Guidelines

This section provides some general guidelines for using the Edge tool:

- For best results, you should use whole-pixel edge detection followed by edge hysteresis thresholding. This produces the edge magnitude image with the most precise edge locations and the fewest spurious edges.
- For image processing applications or methods, use the edge magnitude image.
- For list-based or computationally oriented methods, use the more precise edgelet set.

Using Edge Tool Results

To run the Edge tool you call the global function **cfEdgeDetect()**. Depending on the form you use you can get one or more of the following results:

A magnitude image:
A pel buffer where pixel values represent the found edgelet magnitudes. Edgelets with large magnitudes display as bright spots. This pel buffer is not normally used for display, but is most often used to post process the found edges.

An angle image:
A pel buffer where pixel values represent the found edgelet angles. Edgelets with large angles display as bright spots. This pel buffer is not normally used for display, but is most often used to post process the found edges.

A set of edgelets:
An array of all of the edgelets found by the tool. The edgelets are in no predictable order. See the **ccEdgelet** class reference.

An edgelet index chain list:
An array of chains where each chain identifies a group of edgelets. In general, the Edge tool groups all of the edgelets with the same gradient along a continuous edge, into one chain. For example, all of the edgelets along a circle boundary would form one chain. See the **ccIndexChainList** class reference.

Edgelet Chain Filtering

When you find all of the edges in an image using the Edge tool you often detect many edges not pertinent to your application. Before performing additional processing on the found edges it is best to filter out as many of the unwanted edges as possible. Cognex provides the **cfFilterEdgeletChains()** global function for this purpose.

This global function is designed to accept the array of edgelets and the edgelet index chain list included in the Edge tool result, and run a filter tool on these edgelets to produce a new filtered edgelet index chain list. The tool will accept one or more filters and runs them sequentially, using the filtered output as the input to the next filter. Cognex provides the following filter classes and you can also derive your own filters if you wish.

ccEdgeletChainFilterLength
Provides edgelet chain filtering based on the edgelet chain length. You can set the minimum chain length and if a chain is shorter than the minimum allowed, the entire chain is filtered out. The rationale here is that important edges generally are composed of many edgelets and thus long chains. Noise and unimportant features are often detected as short chains and should be filtered out.

ccEdgeletChainFilterMagnitudeHysteresis
Provides edgelet filtering based on the edgelet magnitude. You can set a low threshold value and a high threshold value and the following rules apply:

- Edgelets with a magnitude less than the low threshold are filtered out.
- Edgelets with a magnitude greater than, or equal to the low threshold are retained if they are in a chain that includes at least one edgelet with a magnitude greater than, or equal to the high threshold.

Note that chains without at least one edgelet magnitude greater than, or equal to the high threshold are completely eliminated.

This filter discards only edgelets and not entire chains. If the filtered edgelet is at either end of the chain, the chain is *trimmed* and a new chain is created. If an edgelet is removed from the interior of a chain, the chain is broken into two new chains.

ccEdgeletChainFilterShape
Provides filtering based on a **ccShape** object, a geometric description. The **ccShape** object should describe a shape you expect to find in run-time images, and describes edges that should be filtered out. This filter could be used in an application where you are analyzing features on a device but wish to ignore the device outline. You specify the device outline as a **ccShape** and the tool will filter out these edges.

If the filtered edgelet is at either end of the chain, the chain is *trimmed* and a new chain is created. If an edgelet is removed from the interior of a chain, the chain is broken into two new chains.

Converting Edgelets to Featurelets

Featurelets and featurelet chain sets provide a way for Cognex vision tools to share a common format for vision tool inputs and outputs. This makes it easier for one application to process results from various vision tools. The Edge tool provides a global function, **cfConvertToFeatures()**, for converting Edge tool results to featurelet chain sets. This conversion, and using featurelet chain sets is discussed in the chapter *Featurelets* on page 19.

Blob

This chapter describes Blob, a vision tool that provides advanced blob analysis of images. This chapter contains the following sections.

The first two sections, this one and *Some Useful Definitions*, give an overview of the chapter and define some terms you will encounter as you read.

Blob Analysis Overview provides an overview of the concepts and constructs that make up blob analysis.

Blob Tool Overview describes the operation of the Blob tool.

Using the Blob Tool describes how to use the Blob tool to analyze scenes, and contains example code showing how to use the Blob tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

featurelet: A common data structure used by several vision tools that describes the sub-pixel location of an image point along with its angle, weight and magnitude. Blob tool results can be converted into featurelet chain sets.

morphology: Process of altering the shapes of regions by applying arithmetic operators to neighborhoods of pixels within an image.

relative thresholding: Defining a threshold value relative to the distribution of pixel values within an image.

run-length encoding (RLE): Data-compression technique that compresses runs of bytes with the same value by storing them as a run length and a value.

second moment of inertia: Tendency of a stationary body to resist rotation about an axis. The moment of inertia of an object about a particular axis is computed by multiplying the mass of each part of the object times the square of the distance of that part from the given axis.

segmentation: Division of the pixels in an image into object pixels and background pixels. Segmentation is an essential part of blob analysis.

thresholding: Segmentation performed by picking a simple threshold value; pixels on one side of the threshold are object pixels, those on the other side are background pixels. Also called *absolute* thresholding.

topology: Method of describing the spatial relationships between regions.

Blob Analysis Overview

The simplest kinds of images that can be used for machine vision are simple two-dimensional shapes or *blobs*. Blob analysis is the detection and analysis of two-dimensional shapes within images.

Blob analysis can provide your application with information about the number, location, shape, and orientation of blobs within an image, and can also provide information about how blobs are topologically related to each other.

When to Choose Blob Analysis

Blob analysis is not appropriate for every vision application. The following characteristics suggest that an application is suitable for blob analysis:

- Images of two-dimensional objects
- High-contrast images

- Presence/absence detection
- Requirements for scale and rotational invariance

On the other hand, applications with the characteristics listed below are probably not suitable for blob analysis:

- Low-contrast images
- Essential features within the scene cannot be represented using just two grey levels
- Requirement for pattern detection

Image Segmentation

Since blob analysis is fundamentally a process of analyzing the shape of a closed object, before blob analysis can be performed on an image, the image must be *segmented* into those pixels that make up the blob being analyzed, and those pixels that are part of the background.

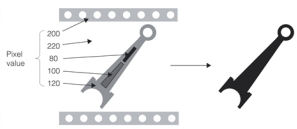
In general, all images that are used for blob analysis start out as grey-scale images of scenes. While it might be easy for a human observer to identify blobs or objects within the scene, before blob analysis can analyze the image, each pixel in the image must be assigned as an object pixel or a background pixel. Typically object pixels are assigned a value of 1 while background pixels are assigned a value of 0.

A number of techniques can be used to segment images into object pixels and background pixels. The following sections describe these techniques.

Hard Thresholding

The simplest technique for segmenting an image is to pick a threshold pixel value. All pixels with grey-scale values below the threshold are assigned as object pixels, while all pixels with values above the threshold are assigned as background pixels. This technique is called *binary thresholding* or *hard thresholding*.

The figure below shows an idealized scene, the application of a hard threshold, and the resulting segmentation of the image into object and background pixels. In the example below, a threshold value of 150 was used. All pixels with values greater than or equal to 150 are treated as background; all pixels with values less than 150 are treated as object.



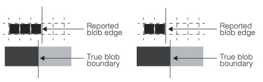
Segmenting an image with a hard threshold

Spatial Quantization Error

Hard thresholding works well at segmenting images. But when blob analysis of the resulting segmented image is performed, the results of the analysis are often degraded by *spatial quantization error*.

Spatial quantization error results from the fact that the exact edge of an object in a scene almost never falls precisely at the boundary between two pixels in an image of that scene. The pixels in which the edge of the blob falls have some intermediate pixel value. Depending on how much of the object lies on the pixel, the pixel is counted as an object pixel or a background pixel. A very small change in the position of a blob can result in a large change in the reported position of the blob edge.

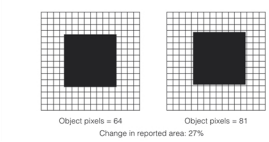
The figure below illustrates spatial quantization error. While the edge of the blob has only moved a tiny fraction of a pixel, the edge of the blob as reported by the hard thresholded image has moved an entire pixel width.



Spatial quantization error

Spatial quantization error can affect the size, perimeter, and location that are reported for a blob. The effect of spatial quantization error on a blob's area differs depending on the shape of the blob.

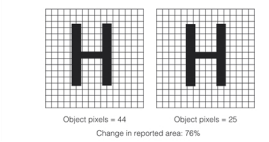
The examples shown in the two figures below assume that pixels covered less than 50 percent by the blob are assigned as background pixels while pixels covered 50 percent or more by the blob are assigned as object pixels.



Object position on a pixel grid affects reported area

In both sets of figures, note how the number of pixels assigned as object pixels changes as a result of a slight shift in the blob's position on the pixel grid. In the figure above, the blob is a square measuring 8.5x8.5 pixels. On the left side of the figure, the blob is centered on the grid so that the blob fully covers 64 pixels (the 8x8 square of pixels at the center of the grid) plus one quarter of each of the pixels that border the blob. Since the border pixels are less than 50% covered by the blob, they are not counted as object pixels and the total number of object pixels is 64. On the right side of the figure the blob is centered on the grid so that the blob fully covers 49 pixels (the 7x7 square of pixels at the center of the grid) plus three quarters of each of the pixels that border the blob. Since the border pixels are more than 50% covered by the blob, they are counted as object pixels, and the total number of object pixels is 81.

The severity of spatial quantization errors depends on the ratio of perimeter to area in the image. Note how much greater the effect of spatial quantization error becomes in the figure below, where there is a greater ratio of perimeter to area.



Spatial quantization error increases with ratio of blob perimeter to blob area

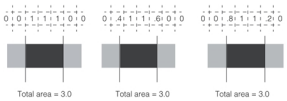
Edges that are aligned with the pixel grid, such as those of rectangles, tend to produce systematic reinforcing errors, while other edges, such as those of round objects, tend to produce random canceling errors.

Soft Thresholding and Pixel Weighting

Spatial quantization error arises in images that have been segmented using a hard threshold because each pixel in the segmented image can assume one of only two values, object (1) or background (0). When the number of possible pixel values in a segmented image increases, pixels in the segmented image can represent the object, the background, or the edge between the object and background.

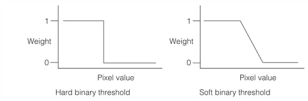
Increasing the number of possible values for pixels in a segmented image is done by assigning *pixel weights*. A pixel weight is a number between 0 and 1.0 that indicates what the pixel represents. A pixel weight of 1.0 means the pixel is part of an object. A pixel weight of 0 means that the pixel is part of the background. A pixel weight between 0 and 1.0 means that the pixel is on the edge of an object.

When blob measures are computed using an image composed of weighted pixel values, measures such as area are computed by summing the pixel weights. When measures are computed based on pixel weights, the effects of spatial quantization error are greatly reduced. The figure below shows a simple 3x1 pixel blob. As the blob moves relative to the pixel grid, the total of the weights of the pixels that contain nonzero pixel values remains constant. This is the case even with the more complex shapes shown in *Object position on a pixel grid affects reported area* on page 216 and *Spatial quantization error increases with ratio of blob perimeter to blob area* on page 216.



Pixel weighting

You convert a grey-scale image into an image segmented into weighted pixel values by supplying a *soft binary threshold*. Unlike a hard binary threshold, which consists of a single threshold value, a soft binary threshold consists of a range of threshold values. Pixels with values above the threshold range are assigned weights of 0 (background), pixels with values below the threshold range are assigned weights of 1 (object), and pixels with values within the threshold range are assigned weights between 0 and 1, typically in a linear manner. The figure below provides a graphical representation of a hard and soft binary threshold.

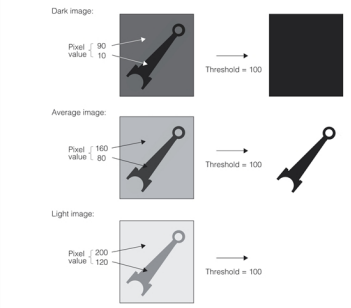


Hard and soft binary thresholds

Note: The thresholds shown in the figure above are appropriate for dark objects on a light background.

Relative Hard Thresholding

The segmentation techniques described in the preceding sections require that you determine the appropriate value for the segmentation threshold. The most common method is trial and error. However, if the images used by your application contain image-to-image variations in brightness, then a static threshold value does not segment the variety of different images correctly. The figure below illustrates this problem.

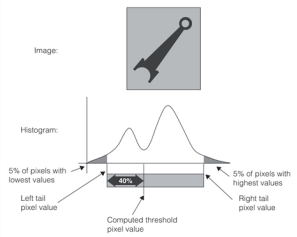


Failure of static threshold value with changing image brightness

The Blob tool allows you to define a threshold value relative to the pixel values within an image.

A relative threshold is specified using two sets of parameters. First, you specify the percentage of the pixels within the image that you want the tool to treat as *tail pixels*. Tail pixels are those pixels within the image that have the lowest and the highest values. In a histogram plot of pixel values, the tail pixels appear at the left and right sides of the histogram. The Blob tool determines the pixel values above and below which the percentages of pixels that you specify lie. These pixel values are called the *right tail pixel value* and the *left tail pixel value*.

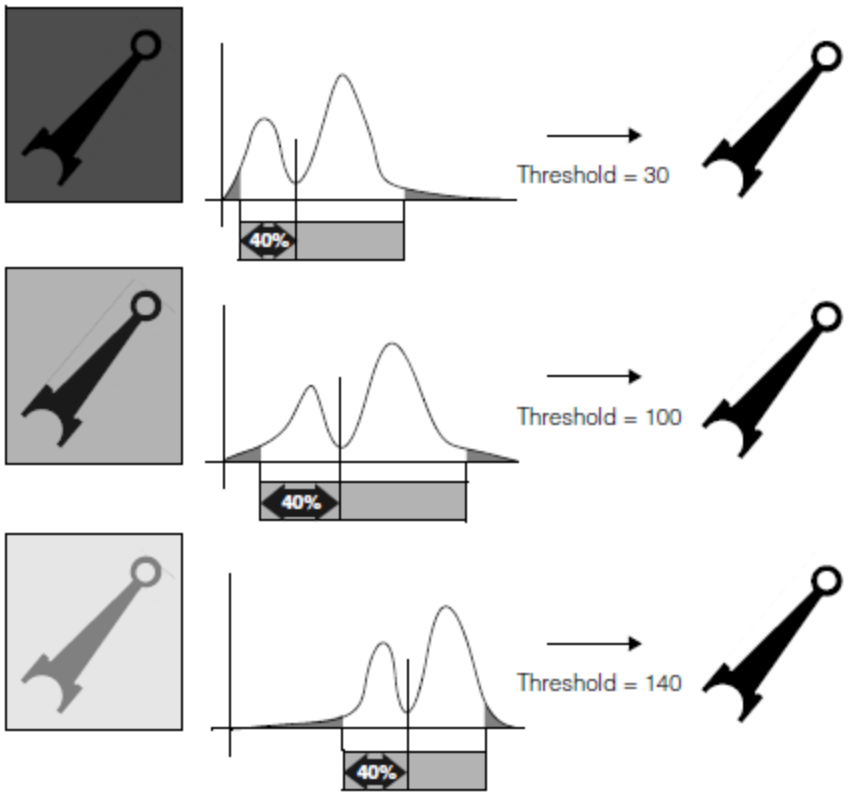
The second parameter you specify for a relative threshold is the threshold percentage. The Blob tool determines the pixel value that lies at the specified percentage of the distance between the two tail pixel values, and the tool uses this pixel value as the threshold pixel value. The figure below shows how the Blob tool computes a relative threshold.



Computing a relative threshold value

In the example shown in the figure above, the left and right tail percentages were specified as 5 percent, and the threshold percentage value was specified as 40 percent. The tool computed the values for the left tail, right tail, and threshold pixel values.

The figure below shows how using a relative threshold can produce an appropriate threshold value for each of the different brightness images.



Relative threshold

While you could perform these steps yourself using histogram analysis and pixel mapping functions, the Blob tool provides the integrated capability to compute a relative threshold value.

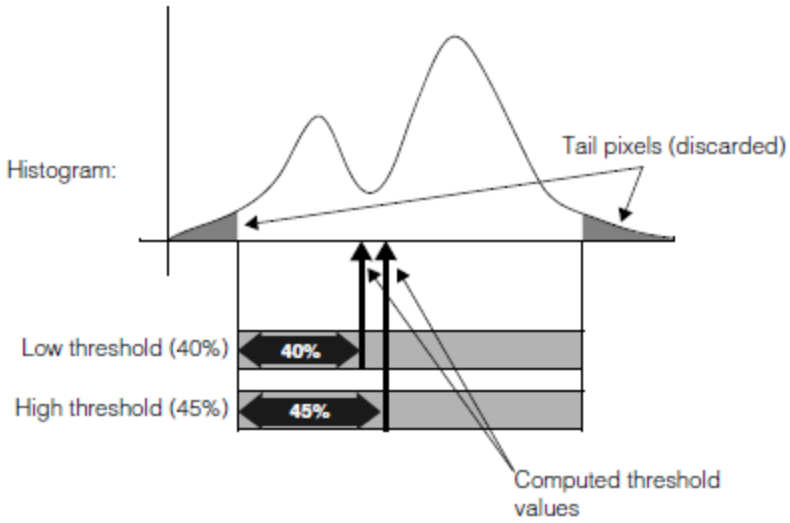
Relative Soft Thresholding

In addition to specifying a hard binary threshold in terms of a percentage of the pixels in the image (after excluding tail pixels), you can also specify a soft threshold using relative values.

To specify a relative soft binary threshold, you need to supply the following values:

- The percentage of low tail and high tail pixels to exclude
- The percentage value to use for the low threshold
- The percentage value to use for the high threshold
- The number of softness steps.

The tool computes a low and high threshold value using the same procedure described in the section *Relative Hard Thresholding* on page 217. The percentage of low and high tail values specified by the tail percentages are discarded and the pixel values the specified percentages of the distance between the remaining low and high values are used as the low and high threshold values, as shown in the figure below.



Computing relative soft threshold values of 40% and 45%

Once the tool has computed the low and high threshold values, it applies a soft binary threshold as described in the section *Soft Thresholding and Pixel Weighting* on page 216.

Pixel Mapping

Certain kinds of scenes cannot be segmented using any kind of binary threshold technique. Binary thresholding only works when all parts of the blob are brighter (or darker) than all parts of the background area. In cases where the blob contains holes and the holes are not the same shade as the rest of the background, binary thresholding always fails.

The figure below shows an example of an image where thresholding does not produce the desired result. The image shows a dark part on a light background. Because of poor lighting conditions, the two holes within the part have a pixel value that is less than the pixel values for the part. Any threshold that assigns the pixels that make up the part as object pixels will also assign the pixels that make up the holes within the part as object pixels. Alternatively, a threshold that assigns the pixels that make up the object to a different segment from the pixels that make up the holes within the part will assign the pixels that make up the part as background pixels. The figure below shows the results of both types of threshold segmentation on the image.

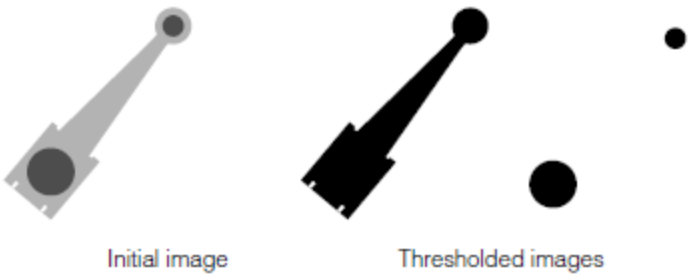
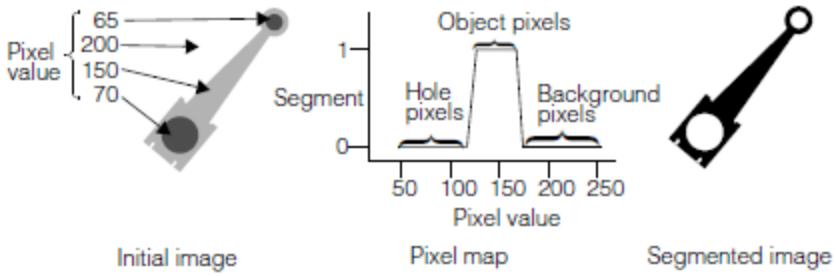


Image that cannot be segmented by thresholding

You can segment images such as that shown in the figure above using a pixel map. A pixel map lets you specify exactly which ranges of pixel values will be assigned as object pixels (pixel weight of 1), which ranges of pixel values will be assigned as background pixels (pixel weight of 0), and which ranges of pixel values will be assigned as edge pixels (pixel weight between 0 and 1). The pixel map is a simple look-up table that defines the pixel weight to assign for each pixel value.

The figure below shows an example of a pixel map that might be used to segment the image shown in the figure above.



Segmenting an image with a pixel map

Threshold Image

Certain types of images resist both thresholding and pixel mapping approaches to segmentation. These images are frequently of scenes that contain a variety of gradient or other lighting problems that result in the presence of the same pixel values in both the object and the background. The figure below shows an example of a scene that cannot be segmented using thresholding or pixel mapping.

Image that cannot be segmented using thresholding or pixel map

Because certain background pixel values are the same as object pixel values, the image cannot be segmented using thresholding or pixel mapping. If you obtain a reference image of the scene where the object is absent, you can segment the image by comparing each pixel in the image to segment with the corresponding pixel in the reference, or threshold, image. Each pixel that differs between the two images by more than a certain amount is assigned as an object pixel. Pixels that are the same between the two images are assigned as background pixels. This technique is known as using a *threshold image*.

Using a threshold image to segment an image involves supplying a threshold image of the scene where the object is absent and a threshold value. When an actual image is segmented, every pixel in the image to be segmented that differs from the corresponding pixel in the threshold image by a supplied threshold value is treated as an object pixel. All pixels that do not differ from the corresponding pixel in the threshold image are treated as background pixels. The figure below illustrates the use of a threshold image.

Segmenting an image using a threshold image

Connectivity Analysis

Once an image has been segmented into object pixels and background pixels, connectivity analysis must be performed to assemble object pixels into connected groups of object pixels or blobs. There are three types of connectivity analysis.

Whole Image Connectivity Analysis

In whole image blob analysis, all the object pixels in the segmented image are treated as making up a single blob. Even if the blob pixels fall into multiple disconnected collections, for the purposes of blob analysis they are considered as a single blob. All the blob statistics and measures are computed using every object pixel in the image.

Connected Blob Analysis

Connected blob analysis uses connectivity criteria to assemble the object pixels within the image into discrete, connected blobs. In general, connectivity analysis is performed by joining all contiguous object pixels together to form blobs. Object pixels that are not contiguous are not considered to be part of the same blob.

Labeled Connectivity Analysis

Depending on the kind of image processing your application performs, you might want to perform blob analysis on images that are segmented into groups of pixels other than object and background pixels. For example, you might segment an image into four different groups of pixels, each group representing a different range of pixel values. This kind of segmentation is called *labeled* segmentation. When you perform connectivity analysis on an image that has undergone labeled segmentation, the connectivity analysis connects all pixels with the same label. With labeled connectivity there is no concept of object and background.

Properties of Blobs

Once an image has been segmented, and the blob or blobs have been located and identified, an application can begin to consider information about the blob or blobs.

A blob is an arbitrary two-dimensional shape. The shape of a blob can be described using a number of different measures. The measures that the Blob tool returns fall into the following categories:

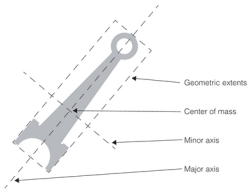
- Geometric properties
- Nongeometric properties
- Topological properties

Each of these groups of properties is described in detail in the following sections.

Geometric Properties

The *geometric* properties of a blob are blob measurements that are intrinsic to the blob itself. Geometric properties can be expressed in relation to a coordinate system defined by the blob itself; as the blob is moved or rotated the properties do not change.

Each of the geometric properties of a blob is described in the following sections. The figure below illustrates the geometric properties of a blob.



Blob geometric properties

Area

The area of a blob is defined as the sum of the weighted pixel values for every nonzero pixel in the blob. If a hard binary threshold was applied to the image to segment it, then the area is simply the number of nonzero pixels in the blob. If grey-scale analysis is being performed, then the area is the sum of the pixel weights.

If the blob is in an image that was segmented using a hard binary threshold, the formula for the area of a blob is $\mathbf{Area} = \sum (x, y)$

If the blob is in an image that was segmented using a soft binary threshold or a pixel map, the formula for the area of a blob is $\mathbf{Area} = \sum W(x, y)$

where W is the weight of the pixel at coordinate (x,y).

Perimeter

The perimeter of a blob is the length of the outside edge of the blob. A blob's perimeter is computed by counting the number of pixel edges between pixels with nonzero values and pixels with values of 0.

Because computing the perimeter in this way tends to produce a perimeter that is larger than the actual blob perimeter, a correction factor is applied to the perimeter.

Center of Mass

The center of mass of a blob represents the blob's balance point. If a sheet of a uniform material were cut out in the shape of the blob, the point upon which the blob would balance is the center of mass. Note that the center of mass of a blob might not actually lie within the blob itself. The figure below shows a blob where the center of mass does not lie within the blob.

Center of mass

The center of mass is computed by determining the average weighted pixel location for each pixel in the blob in the x-axis and y-axis. The formulas that give the x and y components of the center of mass of a blob are

$$C_x = \frac{1}{A}\sum_{x,y} xW(x,y)\ C_y = \frac{1}{A}\sum_{x,y} yW(x,y)$$

where

A is the weighted area.

$W_{(x,y)}$ is the weight of the pixel at coordinate (x,y).

Note: Although the center of mass, as reported in either client or image coordinates, changes as the blob moves, the center of mass with respect to the blob's geometric extents is invariant. For this reason it is described as a geometric property.

Second Moments of Inertia About the Principal Axes

The second moment of inertia of an object about an axis is the tendency of that object to resist being rotated about that axis. If a sheet of a uniform material were cut out in the shape of the blob, the second moment of inertia would indicate how much inertial resistance the object would have to being rotated on the given axis.

The second moment of inertia of a blob is a measure of how widely spread the blob is from a given axis. The higher the value of the second moment, the more widely spread the blob.

The second moment of inertia of a blob about an axis is computed by taking the weight of each pixel, multiplying it by the square of the distance of that pixel from the given axis, then computing the sum of this value for all the pixels in a blob.

The major axis of a blob is defined as the axis about which the second moment of inertia is the smallest. The major axis of a blob is the axis about which it would be the easiest to spin the blob. The major axis of a blob always passes through the blob's center of mass.

The minor axis of a blob is defined as the axis through the center of mass about which the second moment of inertia is the largest. The minor axis of a blob is the axis about which it would be the hardest to spin the blob. The minor axis of a blob is always a line that is at 90° to the major axis passing through the center of mass.

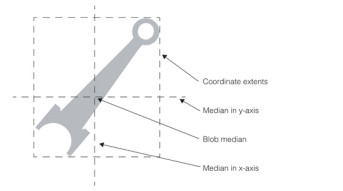
Together, the major and minor axes are referred to as the principal axes.

Geometric Extents

The geometric extents of a blob are the distances between the blob's center of mass and the four sides of a rectangle that is oriented to the blob's principal axes and that completely encloses all the blob's pixels.

Nongeometric Properties

Nongeometric properties of blobs are blob measurements that are meaningful only in relation to the external coordinate system, usually defined by the grid of pixels produced by the frame grabber. As a blob is rotated or changes position, these properties will change as well. Each of the nongeometric properties of a blob is described in the following sections. The figure below illustrates the nongeometric properties of a blob.



Blob nongeometric properties

Blob Median

The median of a blob is the intersection of two lines, one parallel to the x- and the other parallel to the y-axis, such that half of the total pixel weight of the blob lies on either side of the line. Because blob median is a nongeometric property, it is a less reliable indication of a blob's true location than center of mass. However, a blob's center of mass is much more susceptible to spatial quantization errors than is blob median; blob median might be more useful than center of mass in cases where the blob is circular and where a hard binary threshold is being used to segment the image.

Second Moments of Inertia about the Coordinate Axes

The second moments of inertia of a blob about the coordinate axes are measures of how widely spread the blob is relative to the two coordinate axes.

Coordinate Extents

The coordinate extents of a blob are the distances between the blob's center of mass and the four sides of a rectangle oriented to the coordinate axes that completely encloses all the blob's pixels.

Topological Properties

Any scene subjected to blob analysis is ultimately analyzed as a collection of one or more features. A feature can be a blob, or it can be a hole within a blob, or it can be a blob within a hole within a blob. The relationship between blobs and holes in the image is described as the topology of the scene. The principal items of topological information about a feature are

- The feature's label (blob or hole)
- Which features are enclosed by the feature
- Which feature encloses the feature

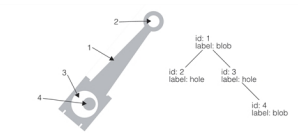
Label

The label of a feature indicates whether the feature is a blob or a hole.

If a labeled connectivity analysis is being performed, a feature's label is the pixel label of each of the pixels that make up that feature.

Hierarchical Position

The figure below shows how the topological relationship among a collection of blobs can be represented using a tree structure. The root blob is that blob that contains all other blobs and holes. Each node on the tree contains a key or ID that identifies the blob; each node on the tree also contains a label indicating whether the feature is a blob or a hole.



Blob topology described using a tree structure

Blob Tool Overview

This section describes how the Cognex Blob tool works, what kinds of operations it supports, and what capabilities it offers.

Image Segmentation

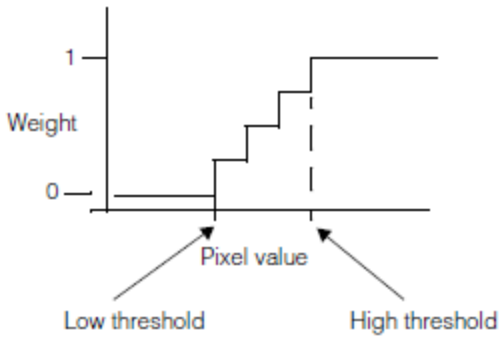
The Blob tool supports each of the segmentation techniques described in the section *Image Segmentation* on page 215.

Hard Binary Thresholding

Support for hard binary thresholding is provided. You can specify the exact pixel value to use as your threshold value and you can define whether pixels above the threshold are to be considered object or background pixels. This allows you to segment correctly images of light objects on a dark background as well as dark objects on a light background.

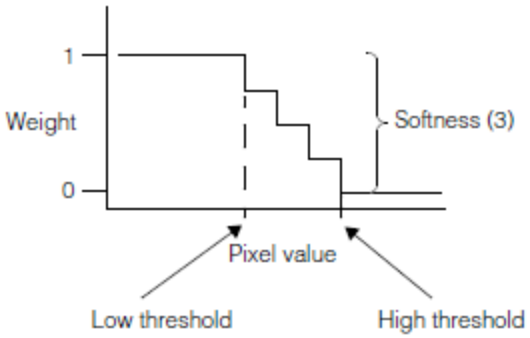
Soft Binary Thresholding

For soft binary thresholding, you supply a low threshold, a high threshold, the softness of the transition (the number of transition levels to use in the threshold range), and an invert flag. These parameters are used to threshold the image, as shown in the figure below.



Soft binary thresholding parameters

The soft binary threshold shown in the figure above would be appropriate for a light object on a dark background. If the threshold were being applied to an image of a dark object on a light background, then the invert flag would be set to true, and the parameters would be interpreted as shown in the figure below.



Soft binary threshold with invert flag set

The actual number of softness levels that the Blob tool uses to segment an input image may be less than the softness value that you specify. The number is related to the distance between the low threshold and high threshold that you specify, but it does not necessarily equal the difference between the low and high thresholds.

Hard Relative Thresholds

To specify a hard relative threshold, you specify the percentage of pixels to treat as left and right tail values along with a threshold percentage. The Blob tool computes the pixel values that define tails of the specified size and then computes the pixel value that lies at the specified percentage of the distance between the two tail values.

Soft Relative Thresholds

To specify a soft relative threshold, you specify the percentage of pixels to treat as left and right tail values along with *two* percentage values, one for the low threshold and one for the high threshold. The Blob tool computes the pixel values that define tails of the specified size and then computes the pixel values that lie at the specified percentages of the distance between the two tail values. These two computed pixel values are used together with a softness value that you specify to construct a soft threshold, as described in the section *Soft Binary Thresholding* on page 224.

Pixel Mapping

For pixel mapping, the tool allows you to specify a 256-element look-up table. The look-up table defines how input pixel values will be assigned pixel weights. Because the table consists of integer values, rather than the floating-point values of 0.0 to 1.0, a scaling factor is also supplied. The scaling factor defines what value in the mapping table (0 to 255) corresponds to a pixel weight of 1. The scaling factor should be equal to the largest value in the look-up table; the pixel weight is equal to the value in the look-up table divided by the scaling factor.

Threshold Input Image

The size of the threshold image must be the same as the size of the image being thresholded. You must also supply a 256-element look-up table that specifies the pixel weight to assign for each difference in pixel value between the threshold image and the image being segmented, and a 256-element look-up table that is used to map the input image before thresholding. In addition to these look-up tables, you must supply a scale value that defines what value in the mapping table (0 to 255) corresponds to a pixel weight of 1.

Previously Segmented Images

The Blob tool also provides support for images that are already segmented. In this case, you supply a scaling value that defines what value in the segmented image (0 to 255) corresponds to a pixel weight of 1.

Image Encoding

Once the segmentation step has been performed, the Blob tool converts the input image into a run-length encoded (RLE) image. Run-length encoding is an encoding technique that takes advantage of the fact that most segmented images contain large numbers of adjacent pixels with the same pixel value. Runs of pixels with the same pixel value are stored as a pixel value and a run length. The resulting RLE image has a number of advantages when compared with an unencoded image:

- Less storage is required for a given image.
- Blob analysis operations can be performed more quickly on an RLE image than on an unencoded image.

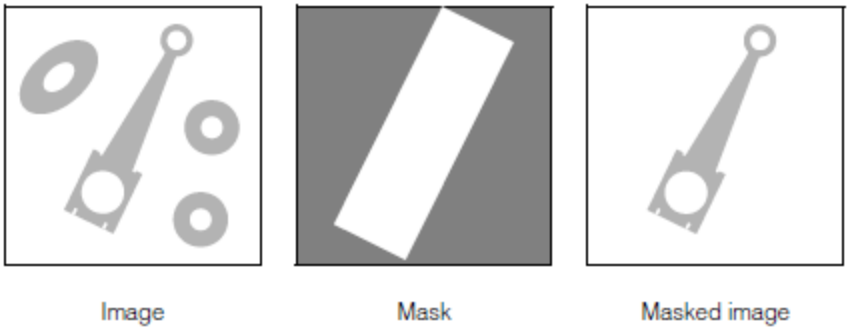
Because RLE images are a compact way of storing an image, and because many blob operations can be performed more quickly using RLE images, the Blob tool provides a facility for obtaining the RLE representation of an image.

Also, a number of image-processing options are provided by the Blob tool. The tool performs these image-processing steps during and after the process of encoding the image as an RLE image.

Masking

You can use an arbitrarily defined mask to exclude portions of an image from blob analysis. To mask an image prior to blob analysis, you must supply an RLE image of the mask. The mask image must have the same dimensions as the image being analyzed; only those pixels in the image being analyzed that correspond to nonzero pixels in the mask image are considered.

You can use masking to remove features from an image that would make segmenting the image difficult or impossible. The figure below shows the use of a mask.



Masking an image before performing blob analysis

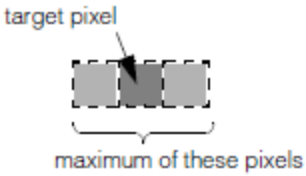
Morphology

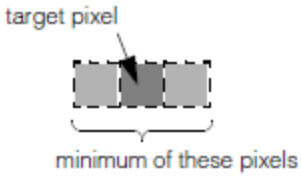
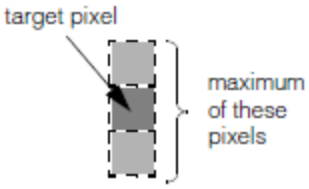
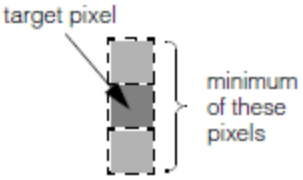
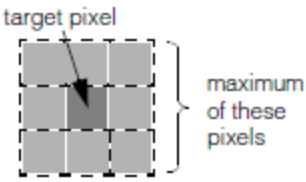
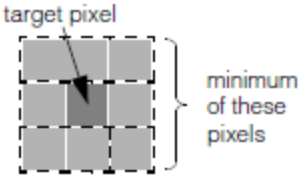
Morphological operations can be applied to images to accentuate or minimize particular types of features within the image. Although morphological operations can be applied to any image at any time, the Blob tool provides integral support for a limited set of morphological operations. Because these operations operate on RLE images, they operate very quickly. This section describes the morphological operations that are available as part of the Blob tool.

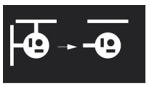
Morphological Operations

Grey-scale morphology is the process of applying a simple operator to successive neighborhoods of pixels within an image. The operators supplied with the Blob tool allow you to substitute for each pixel in an image either the minimum pixel value or the maximum pixel value within a specific neighborhood of that pixel. Each of the neighborhoods and operators is listed in the table below, along with the typical use for the operation, and an idealized representation of the effect of the operation on an image.

Note: Keep in mind that the morphological operators consider *object* and *background* pixels. Which image pixels become object pixels and which become background pixels depends on the segmentation parameters that you specify. The morphological operations always interpret white pixels as object and black pixels as background.

Name	Neighborhood and Operator	Used To	Example
Horizontal dilation (*eMaxH*)	Replace each pixel in the image with the maximum value of the pixel and each of its two horizontal neighbors:  	Reduce or eliminate vertically shaped holes within objects, increasing the thickness of vertically shaped object features.	

Name	Neighborhood and Operator	Used To	Example
Horizontal erosion (*eMinH*)	Replace each pixel in the image with the minimum value of the pixel and each of its two horizontal neighbors: 	Reduce or eliminate vertically shaped object features, increasing the thickness of vertically shaped holes within objects.	
Vertical dilation (*eMaxV*)	Replace each pixel in the image with the maximum value of the pixel and each of its two vertical neighbors: 	Reduce or eliminate horizontally shaped holes within objects, increasing the thickness of horizontally shaped object features.	
Vertical erosion (*eMinV*)	Replace each pixel in the image with the minimum value of the pixel and each of its two vertical neighbors: 	Reduce or eliminate horizontally shaped object features, increasing the thickness of horizontally shaped holes within objects.	
Square dilation (*eMaxS*)	Replace each pixel in the image with the maximum value of the pixel and each of its eight vertical and horizontal neighbors: 	Reduce or eliminate holes within objects, increasing the thickness of object features.	
Square erosion (*eMinS*)	Replace each pixel in the image with the minimum value of the pixel and each of its eight vertical and horizontal neighbors: 	Reduce or eliminate object features, increasing the thickness of holes within objects.	

Name	Neighborhood and Operator	Used To	Example
Horizontal opening (*eMaxMinH*)	A horizontal erosion is applied to the image, followed by a horizontal dilation.	Preserve vertically shaped holes within objects, while eliminating vertically shaped object features.	
Horizontal closing (*eMinMaxH*)	A horizontal dilation is applied to the image, followed by a horizontal erosion.	Preserve vertically shaped features within objects, while eliminating vertically shaped holes within objects.	
Vertical opening (*eMaxMinV*)	A vertical erosion is applied to the image, followed by a vertical dilation.	Preserve horizontally shaped holes within objects, while eliminating horizontally shaped object features.	
Vertical closing (*eMinMaxV*)	A vertical dilation is applied to the image, followed by a vertical erosion.	Preserve horizontally shaped features within objects, while eliminating horizontally shaped holes within objects.	
Square opening (*eMaxMinS*)	A square erosion is applied to the image, followed by a square dilation.	Preserve holes within objects, while eliminating small object features.	
Square closing (*eMinMaxS*)	A square dilation is applied to the image, followed by a square erosion.	Preserve small features within objects, while eliminating holes within objects.	

Morphological operations supported by the Blob tool

Combining Morphological Operations

The Blob tool allows you to combine up to eight morphological operations on an image. The operations are performed sequentially.

Connectivity

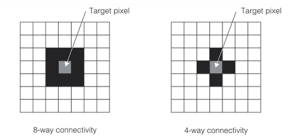
The connectivity phase is performed differently depending on the type of connectivity analysis being performed. The output from the connectivity phase is a *blob scene description* that contains all information about the blobs in the image.

Grey-Scale Connectivity

Regardless of how the image was segmented, the same technique is used to connect pixels into blobs.

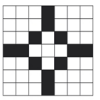
A blob is defined as a group of connected object pixels, where an object pixel is any pixel with a nonzero weight. The Blob tool defines object connectivity in terms of *8-connectedness*; that is, all object pixels bordering a given pixel's edges, as well as those touching its corners, are considered to be connected to that pixel.

Because object pixels are 8-connected, background pixels are 4-connected; that is, background pixels are not considered to be connected diagonally. The figure below illustrates 4-way and 8-way connectivity.



8-way and 4-way connectivity

Defining connectivity in this way has appealing topological properties, as demonstrated in the figure below. In the figure below, if you interpret object pixels (black) as 8-connected, they are considered to be a single blob. If they were 4-connected, there would be eight blobs, because diagonally adjacent pixels would not be considered connected.



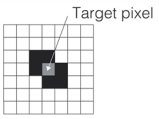
Object pixels are 8-connected, forming a single blob

Interpreting the object pixels in the figure above as a single 8-connected blob is more useful than interpreting them as eight 4-connected blobs.

All features in the image, whether they are blobs or holes, are stored in their own RLE images. These images can then be used for other purposes.

Labeled Connectivity

In the case of labeled connectivity analysis, a blob is defined as a group of connected feature pixels, where each pixel has the same label. Because the division of features into objects and background does not apply to labeled connectivity analysis, the Blob tool performs labeled connectivity in terms of *6-connectedness*. The figure below shows the six pixels that are defined to be connected to a target pixel when the tool performs labeled connectivity analysis.



6-way connectivity

Image Pruning and Filling

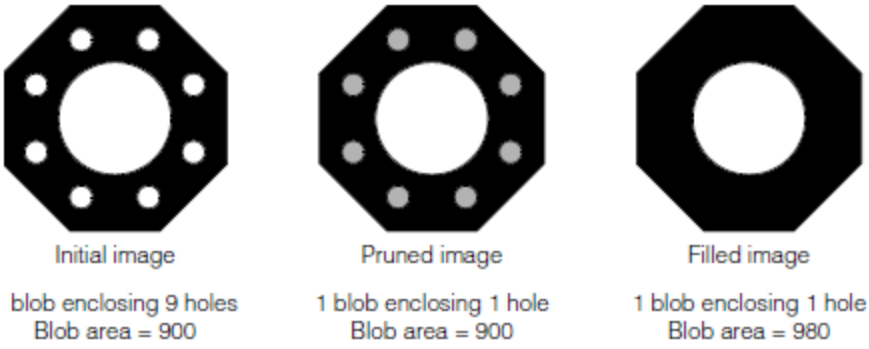
Because of the defects present in most images, such as video noise, even after segmentation and masking, images can contain undesired features. One technique that can be used to remove small features from an image is feature pruning and filling.

During the connectivity phase, the Blob tool allows you to ignore all features (both blobs and holes) below a certain size. This is called image *pruning*. When you prune an image of all features below a certain size, the blob measures returned for the blob that enclosed the pruned features are computed as though the pruned features still existed, but the pruned features themselves are not counted as children of the enclosing feature.

For example, if an image that contains one blob with an area of 900 pixels that contains 8 holes, each with an area of 10 pixels, is pruned using a minimum feature size of 20 pixels, the Blob tool reports the presence of a single blob with an area of 900 pixels.

You can optionally direct the Blob tool to fill in the space occupied by pruned features. This is called *filling* the image. In the case of labeled connectivity, the feature being filled is filled in with pixels with the same label as the enclosing blob. In the case of grey-scale connectivity, the pixel value that is used to fill the feature is the value of the pixel to the immediate left of the feature being filled. As each row of pixels in the feature is filled, the pixel value to the immediate left of that row of pixels is used as the fill value for that row.

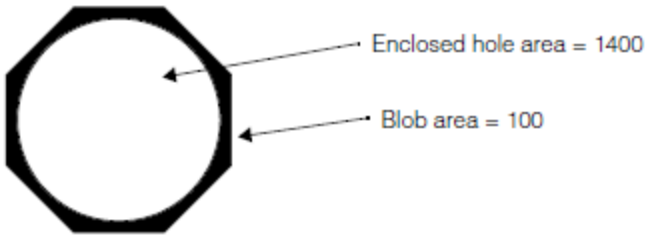
The figure below illustrates the difference between pruning and filling an image.



Pruned and filled images

Keep in mind that the original RLE image is not actually changed by the prune or fill operation; pruning or filling only changes the information that the Blob tool returns about features within the scene. However, if you direct the tool to generate a duplicate image of the scene, or of particular features within the scene, the duplicate image does not contain the pruned or filled features.

Note: If a feature that is small enough to be pruned or filled contains a feature that is not small enough to be pruned, then the enclosing feature is not pruned. The figure below shows an example of this situation. The blob has an area of 100 but encloses a hole with an area of 1400. If a prune size of 200 were used, the blob would not be pruned because it encloses a feature larger than the prune size, even though the blob is smaller than the prune size.



Small blob containing large hole

Blob Analysis

Once the image has been segmented, has been masked, and has undergone morphological operations and connectivity analysis, you can obtain the results of the blob analysis. Blob results are returned in the form of a *blob scene description*. A blob scene description is a data object that provides access to all the information about all features in the scene, including the number of features (blobs and holes), the topological relationship among the features, and the geometric and nongeometric properties of each feature.

In addition, the blob scene description also provides a number of computed blob measures based on the fundamental blob attributes listed above. The specific information contained in a blob scene description is listed below.

Working with a Blob Scene Description

The blob scene description amounts to a collection of chunks of information about blobs. Within the blob scene description each feature (blob or hole) is uniquely identified by an ID. IDs are integers in the range 1 to the number of features in the scene. Certain kinds of information can be obtained regarding the collection of blobs to which the blob scene description refers. Other kinds of information are available for each blob in the scene.

The Blob tool does not compute any information about any feature until you explicitly or implicitly request particular information about that feature. The Blob tool stores all the information about each blob as it computes that information; no measure is computed more than once.

Topological Information

The blob scene description provides access to a tree structure that describes the label and topological relationship of each feature in the scene. The tree structure is depicted in *Blob topology described using a tree structure* on page 223.

The following topological information is available about the blob scene as a whole:

- The number and IDs of all *top-level features*, which are those features that are not enclosed by any other feature in the scene
- The total number of features within the scene

For each feature in the scene the following topological information is available:

- The label of this feature. In the case of grey-scale connectivity, the label is 0 for holes and 1 for blobs. In the case of labeled connectivity, the label is the feature's label.
- The number and IDs of this feature's *child features*, which are those features completely and directly enclosed within this feature
- The ID of this feature's *parent feature*, which is the feature that directly encloses this feature
- The number and IDs of this feature's *sibling features*, which are any other features which are directly and completely enclosed within the same feature that directly encloses this feature

Interior Blobs

You may wish to know if a blob touches the boundary of the current region of interest. This can be useful in the following cases:

- Blobs at the edge of the ROI may represent only part of a blob and therefore blob measurements may be unreliable. These blobs should be discarded.
- You may have a light colored object with dark features, and your background may be the dark feature color. The Blob tool will find the desired features and also may think the background is another blob. Knowing that background blobs touch the ROI edge, you can discard these blobs.

For each blob found by the tool you can determine if it touches the ROI boundary.

Chain Code

For each feature in the scene the blob scene description contains a *chain code* that describes the path around the outside edges of the outermost pixels that make up this feature. The chain code consists of a sequence of direction codes (up, down, left, right) that, when taken as a sequence, describe the perimeter of the feature.

Geometric Information

The geometric measures listed in this section are available for each feature within the scene. For definitions of the geometric measures of features, see the section *Geometric Properties* on page 221.

- Area
- Perimeter
- Center of mass
- Principal axes
- Second moments of inertia about the principal axes
- Geometric extents

Nongeometric Information

The nongeometric measures listed in this section are available for each feature within the scene. For definitions of the nongeometric measures of features, see the section *Nongeometric Properties* on page 223.

- Median about any axis
- Second moments of inertia about any axis through the center of mass
- Coordinate extents about any axis
- Bounding box about any axis

The Blob tool also allows you to compute the size and location of a bounding box oriented with respect to a line drawn at an arbitrary angle through the center of mass.

Computed Information

The following computed measures are available for each feature. These computed measures are derived from the geometric and nongeometric measures listed in the sections *Geometric Properties* on page 221 and *Nongeometric Properties* on page 223.

Acircularity

Two measures of the lack of circularity of a blob are available.

The first is given by the formula $C = \frac{P^2}{4\pi A}$

where

P is the perimeter of the blob.

A is the area of the blob.

The value of C is 1 for a perfectly circular blob. The less circular the blob, the greater the value of C.

The second measure of acircularity is given by the normalized rms deviation of the radius values of the blob from r0, where r0 is the square root of the area of the blob divided by pi.

Elongation

The elongation of the blob is computed by dividing the second moment of inertia about the minor axis by the second moment of inertia about the major axis.

Aspect Ratio

The aspect ratio of the blob is the height of the geometric bounding box divided by the width of the geometric bounding box, where the height is measured along the major axis of the blob and the width is measured along the minor axis of the blob.

You can compute the aspect ratio at any angle.

Working with Nonsquare Pixels

Your frame grabber or camera might have nonsquare pixels. If you are attempting to use the Blob tool to precisely locate and measure objects in the physical world, these nonsquare pixels can make it difficult to associate pixel coordinates with physical coordinates. You can solve this problem by specifying the *aspect ratio* of the pixels that compose the image being analyzed.

Filtering and Sorting Features

Any of the blob measures listed in the preceding three sections can be used to generate a filter through which blobs can be obtained. You can define multiple filters, although only one filter per measure. You can specify that only blobs with values inside or outside a specified range for a specified measure be visible.

You can further specify the order in which the visible blobs are sorted. The sort order determines the order in which the tool returns blobs to you when you request the first, next, previous, or last blob within a particular level of the topological hierarchy.

Navigating a Filtered Topology

If you specify filtering for a blob scene description, then only those features that meet the filtering criteria are visible. As you request child or sibling features, only those features that meet the criteria are returned by the tool.

You can override the filtering when looking at a blob's children by using the filter argument. This allows you to filter blobs, then get detailed, unfiltered information about the features within the blob.

The sort order in effect for a blob scene description affects the order in which child and sibling blobs are returned.

Working with Individual Features

Each feature, blob or hole, within a blob scene description is stored as its own RLE image. You can obtain the RLE image of a particular blob if you want to create or manipulate an image that contains only that particular feature.

Using the Blob Tool

This section tells you how to use the Blob tool to solve specific blob analysis applications.

The basic steps to using blob analysis to solve a vision problem are

1. Obtain images of the scene to be analyzed.
2. Determine the segmentation method and threshold values.
3. Select an image mask (optional).
4. Select morphological operators and pruning and filling parameters (optional).
5. Perform the blob analysis using the parameters selected in steps 1 through 4.
6. Interpret the results of the blob analysis.

Lighting and Optical Considerations

Blob analysis works best with images that can be easily segmented into foreground and background pixels. Typically, strong back lighting of scenes with opaque objects of interest produces images suitable for blob analysis.

Segmenting Images

The process of segmenting an image is the most unpredictable and difficult aspect of blob analysis, particularly if you are confronted with wide image-to-image variation in image contrast and in the number of blobs. This section describes some of the techniques and tools you can use to segment images effectively.

The more consistent images are, and the more information you know about the image at the time you write your application, the more effective will be the segmentation approach.

Choosing a Segmentation Type

The first step is to choose the segmentation type for your application. The table below defines some guidelines for selecting a segmentation type:

Segmentation types compared

Type	Typical Use	Advantages	Disadvantages
Hard binary	Speed-critical applications with large circular objects	Speed	Relatively poor accuracy due to spatial quantization error
Soft binary	Any application with simple binary division between objects and background	Simplicity, accuracy	

Type	Typical Use	Advantages	Disadvantages
Relative	Augments hard or soft binary; useful when brightness changes between images	Handles changing image brightness	Slower than absolute threshold value
Pixel map	Application with non binary division between object and background pixels	Flexibility	Complexity
Threshold image	Application with uneven background illumination where background pixels might have same value as object pixels	Handles otherwise unsegmentable scenes	Requires consistent images

Choosing Segmentation Thresholds

Once you have selected a segmentation type, you must determine the particular threshold value or values your application will use. If the images that your application analyzes have consistent brightness and contrast, you can easily determine an appropriate threshold value by examining several sample images. If your images have significant brightness or contrast variations your application must be able to adjust the threshold value dynamically.

If your application needs to cope with variations in brightness between images, you should consider specifying a relative threshold.

Selecting an Image Mask

You can construct masks synthetically, by drawing an area that you know corresponds to the area of interest within the scene, or you can derive a mask by taking a feature from a blob scene, applying some morphological operations to it, then using it as a mask.

Applying Morphological Operations to an Image

You should select the morphological operations that modify the image in the way that you desire. Remember that morphological operations can change both the geometric properties of blobs such as their size and shape and the topological properties (formerly unconnected blobs might be joined, holes might be opened up, and so on).

Using Nonlinear Client Coordinates

If you supply a run-time image that has a nonlinear client coordinate system, the Blob tool will operate as normal (using the pixels in the image). For each feature (blob or hole) detected in the image, the Blob tool computes the best linear approximation of the nonlinear client coordinate system at the center of mass of the feature. This local linearization is then used to map each of the feature's measurements (center of mass, area, perimeter, and so on) before returning them.

Keep in mind that any local linearization of a nonlinear client coordinate system is most accurate at the point for which the linearization is computed. In the case of a Blob feature, this point is the center of mass. Accordingly, the greatest improvement in Blob tool accuracy offered by this method is for the center of mass. Particularly in the case of larger features (those spanning more than 10 pixels), measures related to the whole feature (perimeter, area, acircularity, and the bounding box-related measures) may tend to show less improvement.

For more information, see the chapter *Using CVL Vision Tools* in the *CVL User's Guide.*

Converting Results to Featurelets

Featurelets and featurelet chain sets provide a way for Cognex vision tools to share a common format for vision tool inputs and outputs. This makes it easier for one application to process results from various vision tools. The Blob tool

provides a global function, **cfConvertToFeatures()**, for converting Blob tool results to featurelet chain sets. This conversion, and using featurelet chain sets is discussed in the chapter *Featurelets* on page 19.

Boundary Tracker Tool

This chapter describes the Boundary Tracker tool, a tool that computes the boundary between an object and the background in an image and returns information about the object such as its area and its orientation in the image.

This chapter contains the following sections:

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

absolute threshold: A single pixel value that divides object pixels from background pixels

angle of a boundary point: The angle of a line tangent to the boundary at that point. When specifying a starting boundary point, the angle should correspond to the initial tracking direction.

boundary: The division between an object and the background in an image

decreasing angle side first: A direction for tracking a boundary where the Boundary Tracker tool tracks the boundary in the direction of the decreasing client coordinate system angle

featurelet: A common data structure used by several vision tools that describes the sub-pixel location of an image point along with its angle, weight and magnitude. Boundary Tracker tool results can be converted into featurelet chain sets.

gradient: A change in pixel values

gradient threshold: The minimum difference that can exist between the candidate pixel and the previous pixel for the pixel to be a boundary pixel

line-drawing algorithm: A method for identifying the set of pixels best suited for representing an arbitrary line segment

threshold: A value that is used to determine whether a pixel is a boundary pixel or not. If the tool is in Absolute mode, then **threshold** is an absolute value that specifies the minimum or maximum pixel value for a boundary pixel depending on the image polarity. If the tool is in Gradient mode, **threshold** specifies the minimum difference that can exist between the candidate pixel and the previous pixel for the pixel to be an object pixel.

Boundary Tracker Tool Overview

The Boundary Tracker tool determines the outline of an object that appears as a silhouette in an image. This silhouette may be either light on a dark background or dark on a light background.

The tool tracks a boundary by either:

- Tracking from a predefined starting point (Track-only mode)
- Searching along a specified number of vectors for a boundary point, and then tracking the boundary from that point. (Search-and-Track mode)

For each boundary that fits specified criteria, the tool returns data which includes:

- A vector of the points in the boundary

  Each point is defined by its subpixel location and (optionally) by its angle.
- The area of the object enclosed by the boundary
- The center of mass of the object
- The x and y-projections derived from the object
- The dominant angle of the object (if you specify that the tool calculate angle data)

The Boundary Tracker tool and the Blob tool both provide information about the shape, location, and orientation of objects in images; however, the Boundary Tracker tool offers two advantages over the Blob tool:

- It is usually faster, sometimes by an order of magnitude, because it processes fewer pixels than the Blob tool.
- It returns the object boundary with subpixel accuracy.

Object Boundaries

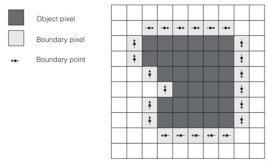
The Boundary Tracker tool requires that your image contain a bimodal distribution of pixel values into object and background pixels. To determine which pixels make up the object, you specify either an absolute threshold or a value that is the minimum difference between an object and a background pixel for the pixel to be an object pixel.

The Boundary Tracker tool returns boundary information as a vector of boundary points. Each boundary point consists of the following information:

- The x- and y-coordinate of the point's subpixel location
- The angle of each point in the boundary (if you specify that angle be computed). The angle returned is the angle of the line tangent to the boundary at the boundary point.

The tool returns one boundary point for each whole-pixel boundary location.

The figure below illustrates the relationship between an object, the whole-pixel boundary, and the boundary points. Each boundary point indicates both the subpixel location of the boundary and the direction of the edge. (The direction of an edge point is tangent to the edge in the tracking direction of the tool.)



Object, boundary pixels, and boundary points

The Boundary Tracking Operation

The Boundary Tracker tool computes the object boundary by first tracking the whole-pixel boundary. Then it uses interpolation to determine the subpixel boundary point locations. The tracking operation has the following stages:

1. The tool locates a point along the object's boundary.

   Either you specify a point on the boundary (Track-only mode), or you specify one or more search vectors along which the tool searches for a boundary point. (Search-and-Track mode)

   For more information about Track-only and Search-and-Track modes, see the section *Specifying a Tracking Mode* on page 243.

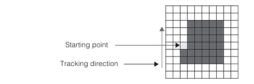
2. The tool tracks the boundary by answering one of the following questions:

   - Is the value of each candidate pixel above (light object on a dark background) or below (dark object on a light background) a single threshold value (Absolute mode)?
   - Is the difference between the value of the previous pixel and the value of the candidate pixel is greater then a preset threshold. (Gradient mode)?

     If the answer is "yes," then the candidate pixel is a boundary pixel.

     For more information about Absolute and Gradient modes, see the section *Specifying a Threshold Mode* on page 242.

The figure below illustrates using the Boundary Tracker tool in Absolute mode to find the boundary of a simple object.

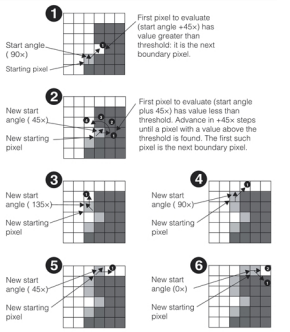


Object, starting point, and initial tracking direction

To track the boundary of a dark object in Absolute mode, the tool starts from a current boundary pixel and evaluates adjacent pixels, starting with the pixel at an angle 45° less than the current angle, to see if each is less than the threshold. The pixels are evaluated in a circular order at 45° intervals until a pixel above the threshold is found.

Once a pixel with a value above the threshold is found, that pixel is the new starting boundary pixel and the starting angle is set to be the angle at which the new boundary pixel was found.

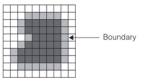
The figure below illustrates how the tool conducts this search. Note that the angles shown in the figure are measured with respect to a typical rotated left-handed coordinate system.



Tracking the object boundary

This boundary tracking algorithm is guaranteed to produce a list of the pixels which enclose an object, given an appropriate threshold value, starting position, and starting angle.

The figure below illustrates the completed boundary track around the object.



Completed boundary track

Once the Boundary Tracker tool has computed the binary boundary, it then computes an interpolated boundary point (and optionally angle) for each binary boundary pixel.

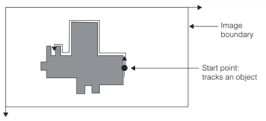
Types of Object Boundaries

This section discusses the following three types of boundaries, which you can track using the Boundary Tracker tool:

- *Objects*, which are closed boundaries which enclose object pixels
- *Holes,* which are closed boundaries which enclose background pixels
- *Open boundaries*, which enclose objects of infinite size

Objects

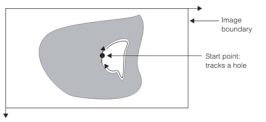
An object is a group of contiguous nonbackground pixels in an image that contains a bimodal distribution of pixels into object and background pixels. Objects are defined to have positive size; so when you specify an area range for an object, you specify it as a range of positive numbers. The figure below illustrates an object.



An object

Holes

A hole consists of a contiguous group of background pixels within an object. The figure below illustrates a hole within an object.



Hole within an object

Holes are defined to have negative size; so when you specify an area range for a hole, you specify it as a range of negative numbers.

Open Boundaries

Open boundaries are defined to enclose objects of infinite size and occur when an object boundary intersects with an image boundary. As described in the section *Open Boundary Conditions* on page 242, you can specify how the tool behaves when the boundary track meets an image boundary.

The figure below illustrates an open boundary.

Open boundary

Open boundaries are defined to have no size. Therefore, when you specify the area range for an open boundary, you specify it as 0 to 0.

Using the Boundary Tracker Tool

To use the Boundary Tracker tool in an application, perform the following steps:

1. Declare a **ccBoundaryTrackerRunParams** object such as *runparams*.

   The class **ccBoundaryTrackerRunParams** has two constructors: one for operating the tool in Track-only mode and a second for operating the tool in Search-and-Track mode. The default constructor constructs an unusable object.

2. Declare a **ccBoundaryTrackerRunResult** object such as *result*.
3. Call the function **cfBoundaryTracker()** to run the tool once you have set up all of the run parameters. This function takes the following arguments:
   - A **const** reference to the image
   - A **const** reference to the run parameters object (*runparams)*
   - A reference to the result parameters object (*result*)
4. Retrieve the data from the result object.

Configuring the Tool

To configure the Boundary Tracker tool, you create a **ccBoundaryTrackerRunParams** object. The constructor that you call to create this object determines whether the tool tracks in Track-only or Search-and-Track mode.

Using the Constructor for Track-Only Mode

When calling the constructor for Track-only mode, you must provide the following information:

1. A starting location along the object boundary
2. The direction in which to track along the boundary
3. The polarity of the object: dark on light or light on dark
4. The following threshold values:
   - A value that separates object from background pixels if in Absolute mode
   - The allowable difference between the value of the current and previous pixels if in Gradient mode. If a pixel is above this gradient, then it is a boundary pixel.
5. The maximum permissible length of the boundary
6. The maximum or minimum area of the object
7. Information about how to compute the angle of each boundary point

Using the Constructor for Search-and-Track Mode

When calling the constructor for Search-and-Track mode, you must provide one or more search vectors in addition to the same data as for Track-only mode.

The following sections provide more information about this data.

Specifying Boundary Criteria

When you create a **ccBoundaryTrackerRunParams** object, you can specify the following parameters that define an acceptable object:

- The maximum number of boundary points for the boundary
- The minimum and maximum sizes for an object

Maximum Number of Boundary Points

You use the maximum number of boundary points to stop the Boundary Tracker tool from tracking after it tracks the specified number of points.

The Boundary Tracker may return up to twice the number of boundary points specified by the *maxBoundaryPoints* run parameter when the following conditions are met:

- Boundary Tracker is run in absolute mode,
- Boundary Tracker is run in search-and-track mode,
- Open boundaries are allowed,
- The tracked boundary reaches the edge of the image, and
- The *stopAtImageEdge* run parameter flag is false.

Minimum and Maximum Object Size

Object size is defined as the total number of pixels inside the object. You can specify a minimum and maximum object size for the object being tracked. If an object is greater or less than the size limit, the tool stops and does not return data about the object if it is in Track-only mode. If the tool is in Search-and-Track mode, the tool searches for another object until it finds one, and does not return data about the object that did not fit within the size criteria.

In all cases, object size is specified in client coordinate system units.

Using Area Range To Specify the Object To Track

The Boundary Tracker tool tracks the boundaries of only those objects whose size falls within a range that you specify. By configuring the tool to track objects of a certain size, you can control the kinds of objects whose boundaries the tool tracks.

Objects have sizes that fall into one of the following regions:

- Hole sizes are negative.
- Sizes of open boundaries are equal to 0.
- Object sizes are positive.

Use the following table to determine the range values that you specify to track objects, holes, or open boundaries:

Region 1: Holes	Region 2: Open Boundaries	Region 3: Objects
areaRange < 0 (Minimum and maximum < 0.)	*areaRange = 0* (Minimum and maximum = 0.)	*areaRange > 0* (Minimum and maximum = 0.)
Track the boundaries of holes only.	Track open boundaries only.	Track the boundaries of objects only.

To track a combination objects, create ranges with overlapping regions. The following table provides examples of different ranges:

Range	Meaning
range [1 to 10000]	Only track closed object boundaries whose size is less than 10000. (Do not track nonhole and open boundaries.)
range [0 to 0]	Track open boundaries only.
range [0 to 5000]	Track open and closed boundaries whose size is less than 5000, but not holes.
range [-100 to 200]	Track holes whose area is smaller than 100, open boundaries, and object boundaries whose area is smaller than 200.

To prevent the tool from tracking holes, specify a minimum object size greater than or equal to zero. To prevent the tracking of open boundaries, specify an area range that does not include 0.

Note: The maximum range value for tracking an object is *DBL_MAX*, which is the default.

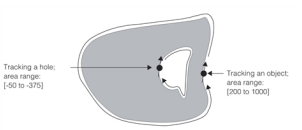
Tracking Several Boundaries Within an Image

The Boundary Tracker tool returns data about one boundary at a time. Therefore, if your application must track the boundary of more than one object in an image, you must call the Boundary Tracker tool once for each object that you need to track.

For example, you may be inspecting parts that consist of an object and a hole. To use the Boundary Tracker tool to track the boundary of both the object and the hole, do the following:

1. To track the object boundary, call the tool with an area range having a positive minimum and maximum.
2. To track the hole boundary, call the tool with an area range having a negative minimum and maximum.

The figure below illustrates tracking both an object and a hole.

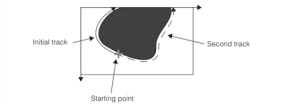


Tracking a hole and an object

Open Boundary Conditions

The Boundary Tracker tool lets you control what happens if the boundary intersects the edge of the image. By default, if the Boundary Tracker tool encounters an image edge while tracking the object boundary, it returns to the starting point and tracks in the opposite direction.

The figure below illustrates this process.



Boundary-image edge collision

You can configure the tool to stop tracking when it encounters an image edge. In this case, the tool only computes the initial track.

Specifying a Threshold Mode

The Boundary Tracker tool uses a threshold value to determine whether a pixel is a boundary pixel or not. This section describes the two threshold modes of the tool: Absolute mode and Gradient mode. How the tool uses the threshold depends on the image polarity, either a dark object on light background or a light object on dark background.

Absolute Mode

In *Absolute* mode, the threshold is a pixel value that you specify. By default, in both Track-only and Search-and-Track modes, the object boundary is defined by a hard threshold value. For dark objects on light backgrounds (the default), all pixels with values below the threshold are object pixels. For light objects on a dark background, object pixels have a value above the threshold value.

When the tool is in Absolute mode, the image is assumed to be binary, having a consistent foreground intensity level that is substantially different from the background intensity level. The quality of the tool's results depends on the degree of separability of foreground and background.

Gradient Mode

While threshold tracking works well for silhouette images, it can fail to produce good results when images have brightness variations. For such images, you can specify that the Boundary Tracker tool use *Gradient* mode.

In Gradient tracking mode, the tool tracks the object boundary by tracking the pixels that exhibit a pixel value difference is greater than a threshold that you specify.

You can use Gradient mode in situations where Absolute mode may not determine the complete boundary of an object. For example, Gradient mode should be used when both the object and the background become progressively darker or lighter. In Absolute mode, the tool would track the object until it reaches an area where the background pixels have values close to object pixels. At this point, the tool might begin tracking background pixels as well as object pixels.

In contrast, the tool in Gradient mode would correctly determine the boundary because the difference between object and background pixels would remain relatively constant.

Gradient Mode Limitations

The Boundary Tracker tool has the following limitations when in Gradient mode:

- It runs only counterclockwise with respect to the object being tracked.
- It does not calculate the following parameters:
  - Area
  - Center of mass
  - The x- and y-projection vectors
- It may not return a closed boundary even for an object that fits entirely within the image.

Specifying a Tracking Mode

The Boundary Tracker tool has two modes for tracking a boundary:

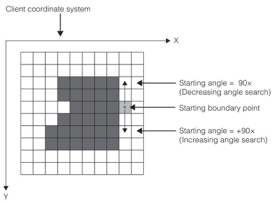
- *Track-only mode*, in which you supply a starting point and tracking direction. The tool tracks the boundary beginning at the starting point.
- *Search-and-Track mode*, in which the tool searches for the object boundary along search vectors that you specify. Once it finds a boundary point, the tool then tracks the object boundary.

Track-Only Mode

For Track-only mode, you specify a starting boundary point that includes the angle of the tracking direction. This angle is the angle of a tangent to the boundary at the boundary point, and corresponds to the tracking direction along the boundary at that point. You specify the search angle when you specify the members of the vector that defines the starting boundary point.

To specify the search direction, set the *decreasingAngleSideFirst* parameter of the **ccBoundaryTrackerRunParams** constructor to false to track the boundary in the direction of increasing client coordinate system angle. Set *decreasingAngleSideFirst* to true to track the boundary in the direction of decreasing client coordinate system angle.

The figure below illustrates how you specify the starting point and direction.



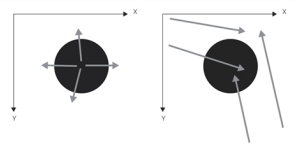
Starting point in track-only mode

Search-and-Track Mode

In Search-and-Track mode, you specify one (or more) search vectors that meet the following requirements:

- The vectors may start either inside or outside of the object and extend through an object boundary.
- The vectors must be approximately perpendicular to the object boundary at the point where they pass through the object boundary.

The figure below shows examples of appropriate vectors to use.



Specifying search vectors for search and track mode

The Boundary Tracker tool locates the edge of the object by searching the pixels that lie along each vector. If it does not find a boundary along one vector, the tool searches the next vector, and so on.

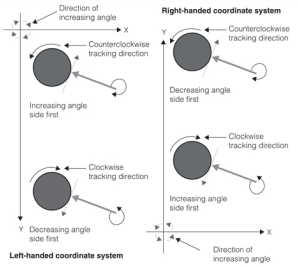
Search-and-Track mode provides the same support for excluding objects and holes based on area that track-only mode does. For a description of using minimum and maximum object size, see the section *Minimum and Maximum Object Size* on page 241.

When the tool is in Track-only mode, it stops tracking if the object being tracked fails to meet the size requirements that you specify. In contrast, when in Search-and-Track mode, the tool continues searching along the search vectors for additional objects until the tool either exhausts the search vectors or finds an object that meets your specifications.

Determining the Tracking Direction

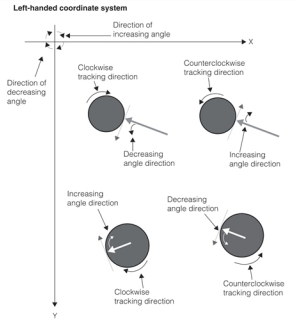
When one of the search vectors intersects a boundary, the tool tracks the boundary in either a clockwise or a counterclockwise direction depending on the setting of the *decreasingAngleSideFirst* parameter of the **ccBoundaryTrackerRunParams** constructor and the handedness of the client coordinate system. If the parameter is set to true, then the tool tracks in the direction of the decreasing angle. If it set to false, the tool tracks in the opposite direction. The *decreasingAngleSideFirst* parameter refers to the direction of the angle created if the search vector were to be rotated in an increasing or decreasing angle direction in the client coordinate system.

The figure below illustrates the relationship among the handedness of the client coordinate system, the setting of the *decreasingAngleSideFirst* parameter, and tool's tracking direction.



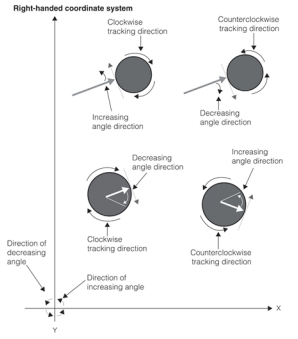
Determining tracking direction from the setting of the decreasingAngleSideFirst parameter

The figure below illustrates the tracking directions for search vectors in a left-handed coordinate system:



Tracking directions in left-handed coordinate system

The figure below illustrates the tracking directions for vectors in a right-handed coordinate system:



Tracking directions in right-handed coordinate system

The table below summarizes the search direction information for the Boundary Tracker tool operating in a left-handed coordinate system:

Start Point	Specified Search Direction	Search Vector Rotation	Tracking Direction
Outside	Increasing angle side		Counterclockwise
Outside	Decreasing angle side		Clockwise
Inside	Increasing angle side		Clockwise
Inside	Decreasing angle side		Counterclockwise

Left-handed coordinate system tracking directions

The table below summarizes the search direction information for the Boundary Tracker tool operating in a right-handed coordinate system. on page 1 summarizes the search direction information for the Boundary Tracker tool operating in a right-handed coordinate system. on page 245 summarizes the search direction information for the Boundary Tracker tool operating in a right-handed coordinate system. on page 245 summarizes the search direction information for the Boundary Tracker tool operating in a right-handed coordinate system.

Start Point	Specified Search Direction	Search Vector Rotation	Tracking Direction
Outside	Increasing angle side		Clockwise
Outside	Decreasing angle side		Counterclockwise
Inside	Increasing angle side		Counterclockwise
Inside	Decreasing angle side		Clockwise

Right-handed coordinate system tracking directions

Data Returned by the Boundary Tracker Tool

The Boundary Tracker tool always returns a vector of boundary points that describes the object boundary. The vector consists of one boundary point vector for each pixel in the binary boundary track. The vector contains the x- and y-coordinates of the point and, optionally, the angle of the point. The tool computes the angle information using a several adjacent boundary points to compute the angle for a given point. You can specify the number of points to use. The more points you specify, the more accurate the angle information.

The tool can also return the following information about the object described by the boundary track:

- The area of the object in client coordinate system units
- The center of mass
- The minimum bounding rectangle aligned along the image coordinate system that encloses the object.

If you specify that the tool compute boundary point angles, the tool can provide information about the x- and y-projections of the portion of the input image contained within the minimum bounding rectangle that encloses the object

Using the Tool in an Application

This section provides some guidelines for using the Boundary Tracker tool in your application.

- Make sure that your image has a bimodal pixel distribution.
- Use histogram analysis to adjust the binary threshold value to account for image-to-image intensity variations.
- If your image exhibits uneven intensity across the image boundary, consider using gradient-mode tracking.
- If you use gradient-mode tracking, specify a maximum boundary size to prevent the tool from tracking indefinitely.

Use Search-and-Track Mode

The Track-only mode requires a starting point that is precisely on the boundary of a shape. A start point that is one pixel off may cause tracking to fail, one that is two or more pixels off will certainly cause tracking to fail. In Search-and-Track mode, on the other hand, you can provide a start point that is near the boundary and a direction in which to search for the boundary, and the tool will track the first boundary that meets the search criteria encountered along this search vector. This is generally a more useful way to use the tool.

Note: When boundary tracker is used in Track-only mode, it may fail to properly close a closed image boundary after it is tracked. It may instead return an open boundary that traverses the entire closed boundary except for a one-pixel gap between the initial and final tracked points. Cognex therefore recommends using the boundary tracker tool in Search-and-Track mode whenever possible.

Converting Results to Featurelets

Featurelets and featurelet chain sets provide a way for Cognex vision tools to share a common format for vision tool inputs and outputs. This makes it easier for one application to process results from various vision tools. The Boundary Tracker tool provides a global function, **cfConvertToFeatures()**, for converting Boundary Tracker tool results to featurelet chain sets. This conversion, and using featurelet chain sets is discussed in the chapter *Featurelets* on page 19.

Ball Pattern Align Tool

This chapter describes the Ball Pattern Align tool, a vision tool that locates a device which has ball pattern and returns the position of the device. It also returns the position, size, and presence of each ball.

Some Useful Definitions on page 247 defines some terms that you will encounter as you read.

What the Ball Pattern Align Tool Does on page 247 provides an introduction to ball pattern alignment.

How the Ball Pattern Align Tool Works on page 249 provides a general description of the operation of the tool.

Using the Ball Pattern Align Tool on page 250 describes the techniques that you use to implement an application using the tool and summarizes the API.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

alignment: The process of running the Ball Pattern Align tool to search a runtime image for matches on a trained ball pattern.

degree of freedom: Part of a transformation that can be characterized by a single numeric value such as angle.

feature: Any specific pattern of grey levels or edges in an image. A feature is a portion of an actual image, as opposed to a model which is an idealized representation of a feature.

granularity: A measure of the minimum detected feature size in an image.

mask: An image used to designate each pixel in a model training image or a runtime image as *care* or *don't care*.

model: An idealized representation of a pattern, stored as a set of grey levels or edges, and containing other data, such as model size and contrast.

model image: Image that contains the pattern you are searching for.

pattern: A trained (internal) geometric description of an object you wish to find in runtime images.

runtime image: Image in which to locate instances of the trained pattern (also called the target image).

transformation: Mathematical representation of the equations that describe the conversion of points from one coordinate system to another coordinate system.

What the Ball Pattern Align Tool Does

The Ball Pattern Align tool locates a device which has ball pattern and returns the position of the device. It also returns the position, size, and presence of each ball. Such a device can be a ball grid array integrated circuit (BGA IC). With the Ball Pattern Align tool, you can verify that such a BGA IC has proper balls before soldering.

To create a model of the ideal ball pattern, the tool can be trained, for example, from an image of a device whose balls are all present and of proper shape and size. In this case, the blob for each ball is extracted and incorporated in the trained model during training. You can also train the tool using circular blobs which you define.

During run time, the alignment of the trained model is performed.

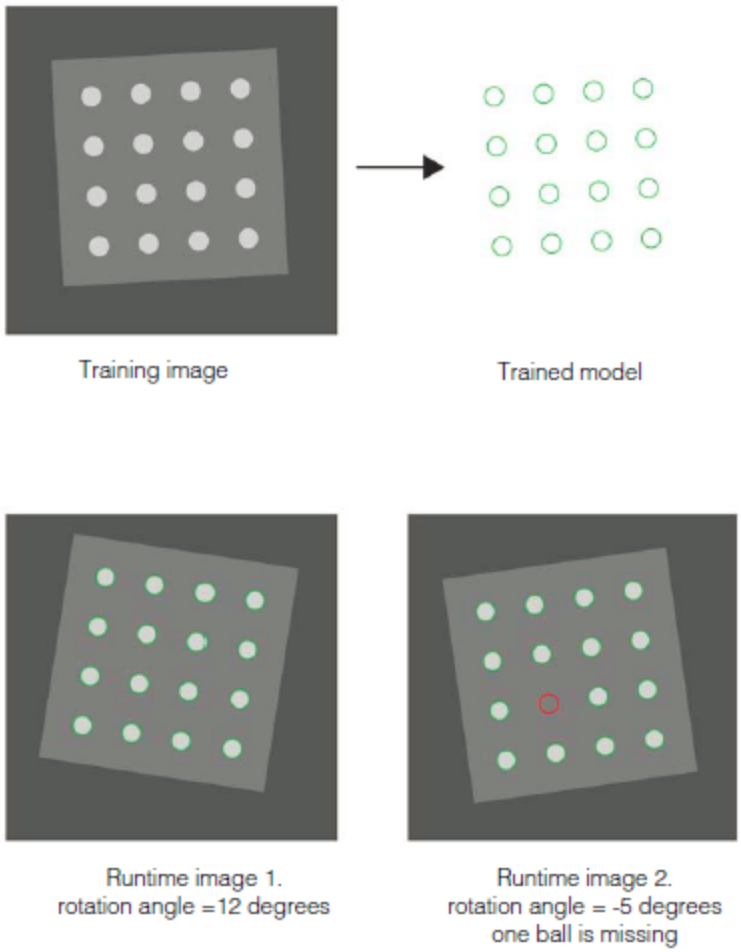
The tool's results include the following:

- Model alignment results:
  - Location of the found model
  - Angle of the found model
  - Scale of the found model

- X scale of the found model
- Y scale of the found model
- Match score of the alignment
- Whether the found model was accepted; that is, the match score was greater or equal to the accept threshold runtime parameter.
- Transform from the training time client coordinates (translated to the model origin) to runtime client coordinates.
- Match region which describes the location of the matched model in the runtime image's client coordinates.
- Vector of ball inspection results for each specified ball

- Ball inspection results for each specified ball:
  - Whether the ball was found
  - Position of the ball
  - Size of the ball

The following figures illustrate model creation and some of the runtime results of the tool. In the runtime results, green outline marks found blobs and red outline marks missing blobs. The rotation angle is the angle of the found model relative to the trained model.



Ball Pattern Align tool train time and run time

How the Ball Pattern Align Tool Works

The Ball Pattern Align tool uses a grey-scale blob to locate the pattern. A grey-scale blob is slightly different from the blob of the Blob tool. A grey-scale blob is identified by its contrast to its immediate background and does not require that a threshold be selectable to binarize the region. Also, it is not based on the connectivity of pels, but rather on sufficient evidence of a feature that is very roughly a circle (including, for example, a square).

A grey-scale blob pattern is a set of such features, such as a ball grid, ball pattern, or either side of a “flip chip” pair (bumps and pads).

The Ball Pattern Align tool is similar to other CVL alignment tools, but is specialized to be robust and fast when facing the unique challenges of training and aligning a pattern of small features.

Also, the Ball Pattern Align tool uses only the blobs for alignment. Other features, such as device bodies, are intentionally ignored.

The interface is intended to be similar to PatMax (refer to *ch_cvl/pmalign.h*), but there are some important differences:

- The Ball Pattern Align tool returns at most the requested number of results, whereas PatMax can return more than the requested number of results if the scores are sufficiently similar.
- The Ball Pattern Align tool allows some additional train-time input to capture your knowledge about the blob sizes.
- The Ball Pattern Align tool requires that the mask pels have values of either 0 meaning *don't care* or 255 meaning *care*. PatMax, by comparison, allows values < 64 to mean *don't care* and values >= 192 to mean *care*.
- As noted earlier, the Ball Pattern Align tool provides the following outputs:
    - pose – a transform from the training time client coordinates (translated to the model origin) to runtime client coordinates.
    - Match score of the alignment

      Outputs derived from these outputs:
    - angle of the found model
    - scale of the found model
    - xScale of the found model
    - yScale of the found model
    - accepted status of the found model
    - matchRegion of the found model

      The following are the similarities to PatMax:
    - The Ball Pattern Align tool allows you to specify a clientFromModel transform so that found poses are specified with respect to a user specified 6 degree of freedom origin (a cc2XformLinear).
    - The Ball Pattern Align tool supports both training time masking and runtime masking, and uses a similar (though more restrictive) convention as does PatMax.

Note: The Ball Pattern Align tool respects timeouts. If a timeout occurs, no valid results are produced by the tool.

Image-based Blob Training versus Geometric (User-supplied) Blob Training

You can train the tool using blobs extracted from an image of an ideal device, called image-based training, or using circular blobs which you define and supply to the tool, called geometric training.

Ball inspection is executed according to the trainedBlobs which were given or generated at train time. A ball which is not listed in trainedBlobs is not inspected. Geometric training allows you to set the ball with numeric data. With non-geometric image-based training, this tool will generate the trained blobs using the supplied training image.

In both cases, ball inspection results include ball position, ball size, and ball presence.

Coordinate Spaces

The Ball Pattern Align tool uses the following coordinate spaces.

Image Space

Image space is a left-handed coordinate system whose origin is at the upper left-hand corner of an image, and whose x- and y-axes are aligned with the image rows and columns, respectively. Distance in image space is measured in pixels (pels).

Physical Space

The physical space is a user-defined real world (x,y) coordinate frame attached to any real structure such as the inspection equipment.

Degrees of Freedom

You can enable the following alterations (degrees of freedom) for the model training and search:

- Angle – enable rotation
- Uniform scale – enable uniform scale
- X scale – enable scaling along the X axis
- Y scale – enable scaling along the Y axis

You can specify the extent of the concrete degree of freedom using the training and runtime parameters of the tool.

Using the Ball Pattern Align Tool

This section describes how to use the Ball Pattern Align tool.

The process of using this tool can be divided into the training-time steps and the runtime alignment steps.

The Ball Pattern Align classes are in *ch_cvl/bpalign.h*

You train a **ccBallPatternAlignModel** using training parameters of type **ccBallPatternAlignTrainParams**. Then you call **ccBallPatternAlignModel::run()** supplying runtime parameters of type **ccBallPatternAlignRunParams**. This function outputs the results as type **ccBallPatternAlignResultSet**. A single result is output as type **ccBallPatternAlignResult**. The tool outputs a result for each model found in the runtime image. For example, if you set **ccBallPatternAlignRunParams::numToFind()** to 2 and there are 2 matching patterns in the runtime image, then you will get two results.

ccBallPatternAlignResult includes the following results:

- **location()** of the found model
- **angle()** of the found model
- **scale()** of the found model
- **xScale()** of the found model
- **yScale()** of the found model
- Match **score()** of the alignment

- Whether the model was **accepted()**
- **matchRegion()**
- **balls()** as a vector of **ccBallPatternAlignBallResult**
- Graphics you **draw()** for the result. You can specify which graphics are drawn using **ccBallPatternAlignDefs::DrawMode**. Graphics include:
    - The origin marking the location of the result.
    - A custom label string marking the origin of the result
    - A bounding box around this result representing the matched region.
    - The trained blobs.
    - The edgelets used for refinement.
    - Found balls in green and not-found balls in red.

      If **accepted()** is true for this result, all graphics, except the edgelets, are drawn in green; otherwise, they are drawn in red.

      A **ccBallPatternAlignBallResult** includes the following results:

        - Whether the ball was **found()**
        - **position()** of the ball
        - **size()** of the ball

          You can choose using a mask during train time and runtime as well. The train-time and runtime masks specify which pixels are *care* and which pixels are *don't care* pixels.

Ball Pattern Alignment Parameter Summary

The following tables summarize the training and runtime parameters of the tool.

Type	Item	Notes
Training parameters	Diameter range	At train time, all balls whose diameter lies within this range are identified. Some or all of these blobs are then trained as the model, depending on the diameter uncertainty.
	Diameter uncertainty	Used to facilitate image-based training, by picking the "most popular" blob size and training only those. The exact effect is to select the largest subset of the found blobs that lie within this uncertainty of their mean diameter.
	Polarity	The target blob polarity.
	Refine mode	Guides the tool how to refine the pose and score at runtime.
	Zone enable	The zone enable for each degree of freedom (DOF). You can enable degrees of freedom to be searched with this.
	Nominal DOF value	The nominal value for the given DOF.
	zoneLow and zoneHigh	Low and high zone parameters for the given DOF.
	Maximum coarse and fine alignment grain limits	Maximum granularity limits for alignment .
	Maximum refine grain limit	Maximum granularity limit for refinement.

Type	Item	Notes
Runtime parameters	Accept threshold	The minimum score for a model to be accepted.
	Numbers to find	The number of results to look for.
	Start pose	The start pose.
	Save match info	Whether to save information necessary to generate a match display.
	Translation uncertainty	The translation uncertainty.
	X-Y overlap	The required X-Y overlap to consider two instances to be the same result.
	Zone enable	The zone enable for each degree of freedom.
	zoneLow and zoneHigh	The low and high zone parameters for the given DOF.
	Zone overlap	The zone overlap parameter for the given DOF.

Ball Pattern Align tool training and runtime parameters summarized

Ball Pattern Align Inspection Typical Usage

The following code shows the typical usage of the tool using image-based (non-geometric) training.

```
//Set Train params
ccBallPatternAlignTrainParams trainParams;
const double ckRatioToleranceToBallSize = 0.3;
double ballSize = 10.0;
trainParams.diameterRange
(ccRange
(ballSize * (1.0 - ckRatioToleranceToBallSize),
ballSize * (1.0 + ckRatioToleranceToBallSize)));
trainParams.polarity(ccBallPatternAlignDefs::eLightOnDark);
trainParams.nominal(ccBallPatternAlignDefs::eAngle, 0);
trainParams.nominal(ccBallPatternAlignDefs::eUniformScale,
 1.0);
trainParams.zoneEnable(ccBallPatternAlignDefs::eAngle);
trainParams.zone(ccBallPatternAlignDefs::eAngle, -45.0, 45.0);
trainParams.maxGrainLimits(16, 8, 2);
//Train BallPatternAlignModel
ccBallPatternAlignModel pattern;
pattern.train(autoImage, trainParams);
// Set Run params
ccBallPatternAlignRunParams runparams;
runparams.acceptThreshold(0.5);
runparams.numToFind(1);
runparams.zoneEnable(ccBallPatternAlignDefs::eAngle);
runparams.zone(ccBallPatternAlignDefs::eAngle, -45.0, 45.0);
// Run BallPatternAlign
```

```
ccBallPatternAlignResultSet rSet;
pattern.run(autoImage, runparams, rSet);
// Get Result
ccBallPatternAlignResult res = rSet.results()[0];
cc2Xform pos = res.pose();
// Get ball Result
bool found = rSet.results()[0].balls()[0].found();
cc2Vect pos = rSet.results()[0].balls()[0].position();
double size = rSet.results()[0].balls()[0].size();
```

CNLSearch

This chapter describes CNLSearch, a vision tool that lets you find patterns in images. It contains the following sections:

Some Useful Definitions on page 254 *defines some terms that you will encounter as you read.*

CNLSearch Overview on page 254 provides an introduction to CNLSearch. It describes some of the concepts that underlie CNLSearch.

How CNLSearch Works on page 257 provides a general description of the operation of CNLSearch.

Using CNLSearch on page 264 describes the techniques that you use to implement an application using CNLSearch.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

edge detection: Process of finding edge pixels in an image.

edge pixel: Pixel in an image that lies on the boundary between two regions of different pixel values.

feature: Any specific pattern of grey levels or edges in an image. A feature is a portion of an actual image, as opposed to a model which is an idealized representation of a feature.

model: An idealized representation of a pattern, stored as a set of grey levels or edges, and containing other data, such as model size and contrast.

model image: Image that contains the pattern you are searching for.

search image: Image in which to locate patterns similar to the model image.

score: Value assigned to a pattern in the search image that measures the similarity between the pattern and the model. Score values are scaled to the range 0.0 to 1.0; the higher the score, the closer the match.

mask: An image used to designate each pixel in a model as *care* or *don't care*.

CNLSearch Overview

The purpose of searching is to locate and measure the quality of one or more previously trained features in an image. Applications for which searching can be useful generally fall into four categories:

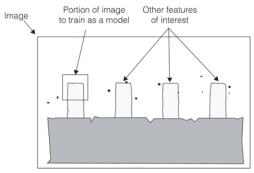
- Alignment: Determine the position and orientation of a known object by locating features (e.g., registration marks) on the object.
- Presence/Absence Detection: Verify that the expected number of features are present in an image.
- Gauging: Measure lengths, diameters, angles, and other critical dimensions for part inspection by locating features in an image, then computing distances between them.
- Defect Detection: Search for defects in an image. Only useful when the appearance of defects is known beforehand.

Features and Models

The search operation measures the extent to which a *feature* in an image matches a previously trained *model* of that feature.

A feature is any specific pattern in an image. A feature can be anything from a simple edge a few pixels in area to a complex pattern tens of thousands of pixels in area. CNLSearch can find features that are defined by a pattern of grey-scale pixel values or by a pattern of edges.

In most cases, you train a representative model from one image and use it to search for similar patterns in that image or in other similar images. The figure below shows an image containing a feature of interest: a lead tip on an electronic component. To train the model of the lead tip, you specify the portion of the image that contains the feature as input to a model training function. CNLSearch creates a model that can be used to search for lead tips in the image from which it was trained or in other similar images.

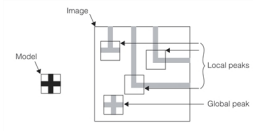


Selecting part of an image as a search model

Search Strategies

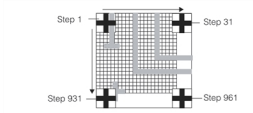
CNLSearch locates features by finding the area of the image to which the model is *most similar*. The term most similar can be used in a *global* sense, meaning the position of greatest similarity (used when looking for a single feature), or in a *local* sense, meaning a position having a degree of similarity that exceeds that of its neighbors (used when looking for multiple instances of a feature).

The figure below shows a model and an image, and the areas of the image that are most similar to the model. An image and model similar to those shown in the figure below might be used to search for a single instance of a feature such as a fiducial mark on a printed circuit board.



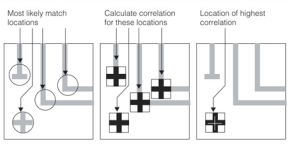
Local and global peaks

A number of strategies can be used to search for a model in an image. The figure below illustrates an exhaustive search method of finding a match for the model. The model is evaluated at every possible location in the image. The location in the image with the greatest similarity is returned. In the figure below, the image is 36 pixels square and the model is 6 pixels square. To locate a match for the model, similarity is assessed at all 961 possible locations.



Exhaustive search

Even with high-performance CPUs, exhaustive search is too slow for most real-world applications. CNLSearch uses a more sophisticated technique for locating features in images. First, the image is quickly scanned to locate positions where a match is likely. Similarity is then assessed only for those candidate locations. The location of the best match is returned. The figure below illustrates this technique.



CNLSearch search technique

Search Score

CNLSearch finds the location of a pattern in a search image based on a model image of that pattern. In addition to returning the location of the pattern in the search image, CNLSearch also indicates how closely the pattern in the search image matches the pattern in the model image by returning a *score*. The score indicates how close a match exists between the trained image and the image whose location was returned. Scores range from 0.0, indicating no similarity between the model and the feature, to 1.0, indicating a perfect match.

In addition to returning the location and score of the closest match found in the search image, CNLSearch can also return the locations and scores of other, less similar instances of the model within the image.

Image Variations: Linear and Nonlinear Brightness Changes

In a typical application, you use CNLSearch to find the location of a particular feature within each of a series of search images. In many applications, the brightness of the image changes between successive search images. These brightness changes are due to a number of factors:

- Changes in illumination intensity
- Changes in illumination source location
- Changes in reflectance of part or all of the scene
- Changes in color of part or all of the scene

The change in brightness that occurs between images can be *linear*, in which case all parts of the image have undergone a proportional change in brightness, or *nonlinear*, in which case some parts of the image have changed in brightness to a different degree than others, or some parts of the image have become brighter while others have become dimmer.

Typically, changing the intensity of light with which a scene is lit causes a linear change in the brightness of different parts of the image. The figure below shows an example of two images with a linear brightness change.

Linear brightness change

Specifically, if the brightness of two images, *j* and *k*, is linearly different, for each pixel value P^{j}_{xy} in image *j,* the value of the corresponding pixel P^{k}_{xy} in image *k* can be calculated using the following formula: $P^{k}_{xy} = A \cdot P^{j}_{xy} + B$

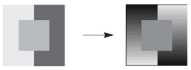
where A and B are constants. If the differences between all corresponding pixel values in the two images cannot be expressed with a linear function of this type, the two images are nonlinearly different.

Nonlinear brightness changes between images are often the result of changes in the reflectance of different parts of the scene. Images of an object obtained at different points during a manufacturing process can undergo nonlinear brightness changes. These changes are a result of process steps that change the surface characteristics of the object. For example, as a metal stamping is painted, the color and reflectance of the stamping changes. Search images obtained at different steps in the painting process can exhibit nonlinear brightness changes with respect to each other and with respect to the model image.

Nonlinear brightness changes in an image can take one of two forms. Different areas within the image may have changed brightness relative to each other while retaining consistent brightness within each area. This is called a uniform nonlinear brightness change. The figure below shows an image that has undergone a uniform nonlinear brightness change.

Uniform nonlinear brightness change

Nonlinear brightness changes within an image can also occur within formerly uniform areas of the image. This is called a nonuniform nonlinear brightness change. The figure below shows an image that has undergone a nonuniform nonlinear brightness change.



Nonuniform nonlinear brightness change

CNLSearch can find patterns in search images with both uniform and nonuniform nonlinear brightness changes from the model image.

How CNLSearch Works

When you use CNLSearch to search for a pattern in an image, you must specify whether the search image is linearly or nonlinearly different in brightness from the model image. If you specify that the search image is linearly different, CNLSearch operates in *linear* mode; if you specify that the search image is nonlinearly different, CNLSearch operates in *nonlinear* mode.

Nonlinear mode searches work with both linear and nonlinear brightness changes between the model image and the search image, whereas linear mode searches only work with linear brightness changes between the model image and the search image. Nonlinear mode search tends to be less accurate and take longer than linear mode search, however.

CNLSearch computes the search score differently depending on whether CNLSearch is used in linear mode or nonlinear mode. In addition, in nonlinear mode CNLSearch computes two *component scores* that describe different aspects of the similarity between the model image and the search image. The overall score is based on the component scores.

The next two sections describe the operation of CNLSearch in both linear and nonlinear modes. For information on how to choose between linear mode and nonlinear mode for your application, see the section *Choosing Between Linear and Nonlinear Mode* on page 268.

Searching in Linear Mode

If you specify that the search image is linearly different in brightness from the model image, CNLSearch finds the part of the search image where the pattern of pixel values is the most similar to the pattern of pixel values in the model image. This type of searching is called *intensity correlation* searching because the degree of similarity between the search image and the model image is determined by calculating the correlation coefficient between the patterns of grey-scale pixel values in the two images.

The method that CNLSearch uses to compute the correlation coefficient between the two images is not affected by linear changes in brightness between the images.

Searching in Nonlinear Mode

If you specify that the search image is nonlinearly different in brightness from the model image, CNLSearch searches for the model within the image by finding the location within the search image where the pattern of edges and non-edges is the most similar to the pattern of edges and non-edges in the model image.

Because CNLSearch's nonlinear mode searches for patterns of edges instead of patterns of pixel values, CNLSearch is immune to both linear and nonlinear brightness changes between the model image and the search image, as long as the brightness changes do not affect the pattern of edges in the search image.

Search Parameters

When you perform a search using CNLSearch, you specify parameters that CNLSearch uses to determine whether a particular feature within the image is a valid instance of the model. In both linear mode and nonlinear mode, you specify the following parameters for the search:

- The *acceptance threshold*

  The acceptance threshold is the score (between 0.0 and 1.0) that CNLSearch uses to determine whether a match represents a valid instance of the model within the search image. Matches with nonzero scores greater than or equal to the acceptance threshold are valid matches.

  You use the acceptance threshold to indicate to CNLSearch the degree of image degradation that you expect in search images. If you expect search images to be degraded, you should specify a lower acceptance threshold.

- The number of instances of the model you expect the search image to contain

  CNLSearch returns the location of every instance of the model within the search image that has a score that exceeds the acceptance threshold, up to the number of instances you specify. If there are fewer instances of the model with scores above the acceptance threshold than you specify, CNLSearch might return the location and score of some additional instances with scores below the acceptance threshold, although those instances will be marked as not found.

- The *confusion threshold*

  The confusion threshold is the score (between 0.0 and 1.0) that represents the highest score that a feature that is *not* an actual instance of the model will receive. You should always set the confusion threshold greater than or equal to the acceptance threshold.

  CNLSearch treats the confusion threshold you specify as a hint about the nature of the image being searched. You should specify a high confusion threshold to indicate that the scene contains features that resemble the model but are not valid instances of the model; specify a low confusion threshold to indicate that the only features in the scene that resemble the model are valid instances of the model.

  You use the confusion threshold to indicate the difficulty you expect CNLSearch to have discriminating between valid instances of the model and other features in the image. If you expect CNLSearch to have difficulty discriminating which are valid instances of the model, you should specify a higher confusion threshold.

  In general, a higher confusion threshold can increase the reliability of searches at the cost of somewhat slower searching. A properly selected confusion threshold lets CNLSearch provide the best balance of reliability and speed.

- The *method*

  The search method specifies the level of accuracy of the search. CNLSearch supports *coarse*, *fine*, and *veryfine* searches. More accurate search methods take more time and require more memory than less accurate search methods.

Linear Searches

When you perform a search using CNLSearch in linear mode, it returns the location of the part of the search image with pixel values that are the most closely correlated to the pixel values in the model image.

Linear Mode Search Algorithms

CNLSearch lets you choose between two algorithms to perform linear mode searches. These algorithms use different search strategies. For most applications, the **CnlpasLinear** algorithm is the best choice. It uses a relatively conservative strategy for identifying likely matches within the image. This strategy is somewhat time consuming, but it greatly reduces the risk of missing an actual instance of the model in the image.

If your application needs maximum speed, you can use the **Search** algorithm. This algorithm takes a more aggressive approach to locating likely matches. Because of this, it may tend to discard some unpromising locations prematurely.

Correlation Searching

The *correlation coefficient* of one pattern of pixel values to another is expressed as a number between -1.0 and 1.0. A correlation coefficient of 1.0 means that the pixel values in the two images are perfectly matched. A correlation coefficient of -1.0 means that the pixel values in the two images are perfectly mismatched. A correlation coefficient of 0 means that pixel values in the two images are randomly different.

The figure below shows three sets of image pairs, one pair with a positive correlation (a), one pair with a negative correlation (b), and one pair with an insignificant correlation (c).

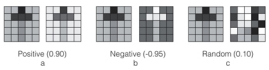


Image pairs showing different correlation coefficients

The figure below shows the effect that a nonlinear brightness change has on the correlation coefficient of a pair of images. While the pattern is still recognizable, the correlation coefficient is little better than that of the random image pair shown in the figure above.

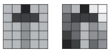


Image pair with weak positive correlation coefficient (0.15) due to nonlinear brightness change.

Computing the Correlation Coefficient

Mathematically, the correlation coefficient r of a model and a corresponding portion of an image at image offset (u,v) is given by

$$r\left(u,v\right)=\frac{[N\sum_i I_i M_i-(\sum_i I_i)\,(\sum_i M_i)]}{\sqrt{\left[N\sum_i I_i^2-(\sum_i I_i)^2\right]\left[N\sum_i M_i^2-(\sum_i M_i)^2\right]}}$$

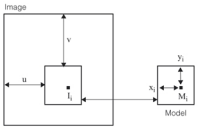
where

N is the total number of pixels.

I_i is the value of the image pixel at $(u+x_i, v+y_i)$.

M_i is the value of the corresponding model pixel at the relative offset (x_i, y_i).

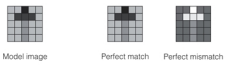
The figure below shows the relationship among these components.



A model and the corresponding portion of the image

The value of r is always in the range -1.0 to 1.0, inclusive. A value of 1.0 signifies a *perfect match* between the area of the image and the model. Specifically, if r = 1.0, there exist some values a and b such that for all i: $I_i = aM_i + b, a > 0$

A value of -1.0 signifies a *perfect mismatch*, which is the same as a perfect match except that a < 0: the feature found is the *negative* of the model. The negative of a model or image is a corresponding image in which the sense of light and dark has been reversed. The figure below shows an example of a perfect match and perfect mismatch.



Perfect match and perfect mismatch

CNLSearch lets you treat mismatches as if they were matches.

Score for Linear Mode Searches

CNLSearch computes a score for each instance of the model that it finds in the search image. CNLSearch has two methods for computing scores. *Absolute* and *normalized* scoring. The table below describes how the various scores are computed. In all cases, the scores are derived from the correlation coefficient between the pixel values in the model and the pixel values in the image, as described in the section *Computing the Correlation Coefficient* on page 259.

Scoring Method		
Normalized	If the correlation coefficient is less than 0, the score is 0.0. Otherwise, the score is equal to the correlation coefficient squared.	Treats mismatches as if they received a score of 0
Absolute	Correlation coefficient squared	Treats mismatches as if they were matches

Linear mode scores

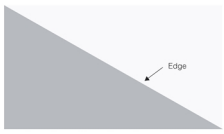
For more information, see the section *Choosing Between Normalized and Absolute Scoring* on page 271.

Nonlinear Searches

When you perform a search using CNLSearch in nonlinear mode, it returns the location of the part of the search region with the pattern of edges that most closely resembles the pattern of edges in the model image.

Edge Detection

Within an image, the boundary between two regions with different pixel values is called an *edge*. An edge has a magnitude which is related to the difference in pixel values for pixels on either side of the edge. The figure below shows an edge.

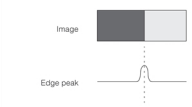


An edge

A pixel in an image that lies on an edge in the image is called an *edge pixel*. The process of determining which pixels in an image are edge pixels is called *edge detection*.

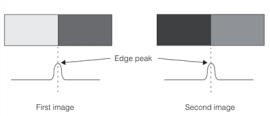
In order to be considered to be an edge pixel by CNLSearch, a pixel must be one of a group of connected pixels where the average difference in pixel values between the regions on either side of the edge is greater than an edge threshold that you can specify.

The figure below shows an idealized representation of an edge along with a graph showing the edge peak.



Locating an edge.

The only information contained in images that have undergone nonlinear brightness changes that does not change as a result of the brightness changes is the location of edges within the image. The figure below shows how the edge peak occurs in the same location despite a nonlinear change in brightness between the images.



Edge detection with nonlinear brightness change

Edge Maps

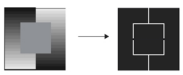
For each pixel in an image, CNLSearch determines the edge magnitude of that pixel. CNLSearch performs edge detection in both the model image and the search image and creates *edge maps* of both images.

The figure below shows a grey-scale image and an edge map created from that image.

Converting a grey-scale image into an edge map

The figure below shows an edge map created from an image with a nonlinear brightness change from the source image used in the figure above. Note that the edge maps are very similar, despite the nonlinear brightness change between the source images.

CNLSearch lets you generate edge maps of any image.



An edge map generated from an image with a nonlinear brightness change

Edge Threshold

When you train a model and when you perform a search using CNLSearch in nonlinear mode, you specify a pair of edge thresholds. The edge thresholds set the edge strength (expressed as the difference in pixel values across the edge) that CNLSearch uses to identify an edge.

You specify a *low threshold* and a *high threshold*. All edges with strengths above the high threshold are included in the edge map. All edges with strengths below the low threshold are excluded from the edge map. Edges with strengths between the thresholds are included in the edge map if they are *8-connected* to another edge with a strength above the threshold, either directly or through other edges with strengths between the thresholds.

Edge Sharpness

The edges in an image can be sharp, in which case the change in pixel value that defines an edge takes place across a small number of pixels, or the edges in an image can be fuzzy, in which case the change in pixel value that defines an edge takes place across a large number of pixels. The figure below shows examples of sharp and fuzzy edges.

Sharp and fuzzy edges

Nonlinear mode CNLSearch searches produce more accurate results when the search and model images have sharp edges.

Searching Edge Maps

When you perform a search using CNLSearch in nonlinear mode, it locates the part of the search image's edge map that is the most closely correlated to the model image's edge map. The correlation is computed using a formula similar to that used to compute correlation coefficients in linear mode, but instead of computing the correlation coefficient of the grey-scale images, CNLSearch computes the correlation coefficient of the edge maps.

An important consequence of this method of computing the correlation coefficient of a pair of edge maps is that both missing and extraneous edge pixels in a search image have an effect on the resulting score. Also, since the correlation coefficient is computed based on the entire area contained within the model, models should contain as small a proportion of non-edge pixels as possible.

Score for Nonlinear Mode Searches

The score for Nonlinear mode search is computed in the same way as for linear mode searches, except that instead of comparing pixel values between the model image and the search image, CNLSearch compares the model image edge map and the search image edge map.

In addition to the overall score, CNLSearch also computes a pair of component scores.

- The *area* score considers the edge pixels in the entire area of the search image that corresponds to the model image. The area score is based on the correlation coefficient between the entire area of the model image edge map and the search image edge map.
- The *edge* score considers on the pixels in the search image that correspond to edge pixels in the model image. The edge score is the percentage

The table below summarizes how CNLSearch computes scores in nonlinear mode.

Area Score	Edge Score	Overall Score
If the correlation coefficient is less than 0, the score is 0.0. Otherwise the score is equal to the correlation coefficient squared.	The percentage of edge pixels from the model image that are also present in the search image mapped to the range 0.0 through 1.0	Area score

Nonlinear mode scores

Note that nonlinear mode search does not support absolute scoring. All mismatches are assigned a score of 0.0.

Occlusion and Clutter

When CNLSearch computes the area score in nonlinear mode, it considers the entire area of the model image. Every non-edge pixel in the model image that is also a non-edge pixel in the search image is counted as a matching pixel and increases the area score. Every edge pixel in the model image that is also an edge pixel in the search image is counted as a matching pixel and increases the area score.

Mismatches between the model image edges and the search image edges fall into two categories.

- Model edge pixels that are not present in the search image. These are called *occlusions*.
- Search image edge pixels that are not present in the model image. These are called *clutter*.

By default, when CNLSearch computes the score for a match, it weights occlusion more heavily than clutter when it computes the area score in nonlinear mode. This tends to increase the difference between the scores received by actual instances of the model, which might be degraded by image defects and noise, and other parts of the image that might be confused with actual instances. This weighting makes it easier for your application to distinguish actual instances of the model from other, confusing parts of the search image.

Using Area Score and Edge Score in Nonlinear Mode Searches

Because the area score for nonlinear mode searches is affected by both missing edge pixels and extraneous edge pixels in the search image, and because the overall score for a nonlinear search is the same as the area score, a low score for a nonlinear search can be caused by any of the following situations:

- If the search image has such low contrast that CNLSearch cannot reliably detect edges within the image, CNLSearch returns a somewhat reduced area score and a low edge score.
- If the search image is a poor match for the model image, CNLSearch returns both a low area score and a low edge score.
- If the search image is a good match for the model image, but the search image contains extraneous edges, then CNLSearch returns a high edge score and a low area score.

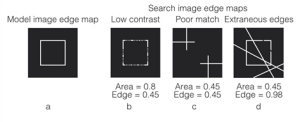
Your application can distinguish between these three causes by comparing the area score with the edge score returned for a search. This technique is illustrated in the figure below.

The figure below a shows an edge map generated from a model image. The figure below, items b-d show edge maps generated from three search images. The first search image edge map (b) shows an image that had very low contrast. The second search image edge map (c) shows a poorly matched image. The third search image edge map (d) shows a well matched image that contains extraneous edge pixels.

The first search image (b) has a high area score, because almost all of the non-edge pixels in the model image correspond to non-edge pixels in the search image. The image has a low edge score, indicating a lack of similarity between the edges in the model image and the search image.

The second search image (c) has a low area score and a low edge score, indicating a lack of similarity both between the entire area of the edge maps and between the edges in the model image and the edges in the search image.

The third search image (d) has a low area score but a high edge score, indicating that the edges in the model image are also present in the search image, but that when the entire area of the image is considered, the model image edge map is not very similar to the search image edge map.



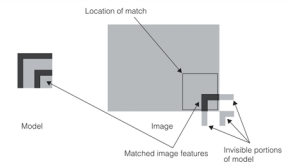
Area score and edge score for nonlinear search

By considering both the area score and the edge score for nonlinear searches, your application can distinguish between low scores caused by low contrast images, low scores caused by poorly matched images, and low scores caused by extraneous edge pixels.

Repeating Pattern Models

If the model image contains a repeating pattern of edges and the search image contains a partial instance of the model, CNLSearch might return a poor area and edge score. In this case, the position reported for the model instance is inaccurate.

The figure below illustrates an example of this condition. An instance of the model is located partially outside the image area. The upper left corner of the model instance in the search image matches the lower right corner of the model image. CNLSearch returns a low score for the instance because only part of the model matches, but the location returned does not represent the actual location of the model in the image.



False match of repeating pattern model

Comparing Linear and Nonlinear Mode Scores

Although CNLSearch returns a score in the range between 0.0 and 1.0 when used in both linear mode and nonlinear mode, scores for linear searches are not directly comparable with scores for nonlinear searches. Scores can only be compared if they are the results of searches performed using the same mode.

In general, scores are more variable for nonlinear mode searches than for linear mode searches.

Note that scores for both linear mode algorithms *are* comparable with each other, since the same formula is used to compute the score.

Finding Features at the Edge of the Image

When CNLSearch finds a feature at the edge of the search image, the accuracy with which it can report the position of the feature is reduced. When CNLSearch returns result information, it includes information about whether or not the feature was found at the edge of the image. You can use this information to make sure that you interpret the position information for different matches correctly.

Using CNLSearch

This section describes how to use CNLSearch.

Overview of Using CNLSearch

The process of using CNLSearch can be divided into the training-time steps and the search-time steps.

In general, you should follow these training-time steps:

1. Choose between linear and nonlinear mode search. If you are not sure which mode your application requires, you can train a model for both modes.
2. If you are using linear mode search, select your algorithm. You can train the model for both linear mode algorithms.
3. Select the search accuracy level. You can train the model for any or all accuracy levels.
4. Select a model image and train the model.

In general, you should follow these search-time steps:

1. Perform a series of test searches at different accuracy levels and with different algorithms.
2. Fine-tune the various search parameters, including the acceptance and confusion thresholds using the results of the test searches.
3. (Optional) Re-train the model using just the algorithms and accuracies that work best for your application. This can reduce the amount of memory required to store the model, but it does not change the speed of subsequent searches.

The table below contains an overview of the different training-time and run-time parameters you supply to CNLSearch.

	Parameter	Values	Notes
Training	Accuracy	• Coarse • Fine • Veryfine	You must train the model for all accuracy levels you intend to search with. Training for additional accuracy levels increases training time and the amount of memory required for the model.
	Algorithm	• CnlpasLinear • CnlpasNonlinear • Search	You must train the model for all the algorithms you intend to search with. Training for additional algorithms increases training time and the amount of memory required for the model.
		• Normalized • Absolute	Training for both normalized and absolute scoring has no cost at training time

	Parameter	Values	Notes
Search	Accuracy	• Coarse • Fine • Veryfine	Specify the search accuracy. The greater the accuracy, the slower the search. For more information, see the section *Selecting a Search Accuracy* on page 270.
	Algorithm	• CnlpasLinear • CnlpasNonlinear • Search	Use the guidelines in the section *Choosing Between Linear and Nonlinear Mode* on page 268.
		• Normalized • Absolute	Use the guidelines in the section *Choosing Between Normalized and Absolute Scoring* on page 271.
	Confusion Threshold	0.0 - 1.0	The best score that a non-instance of the model can receive. For more information, see the section *Selecting a Confusion Threshold and an Acceptance Threshold* on page 268.
	Acceptance Threshold	0.0 - 1.0	The worst score that an instance of the model can receive. For more information, see the section *Selecting a Confusion Threshold and an Acceptance Threshold* on page 268

CNLSearch parameter overview

Training a Model

When you train a model using CNLSearch, you specify the types of searches to be performed using the model. CNLSearch generates and stores only the information about the model needed to perform the types of searches that you specify. You can specify that a model be trained to perform all types of searches, in which case CNLSearch generates and stores all the information required to perform all types of searches.

Selecting the Model Image

Observing the following guidelines will help you select an effective model image.

- If you plan to train the model for both linear mode and nonlinear mode searches, you should select a model that includes both grey-scale pattern information and edge information.
- Your model should include a balance of both strong horizontal and vertical elements; avoid a model that has all horizontal or all vertical features. Selecting a model that is roughly square can help achieve a balance of horizontal and vertical features.
- Select models that contain as much *redundancy* as possible. A redundant model contains enough elements that there will be something to match even if the model is partially obscured in the search image.

The figure below shows some examples of models that are roughly square, contain a balance of vertical and horizontal elements, and contain redundant features.

Good models

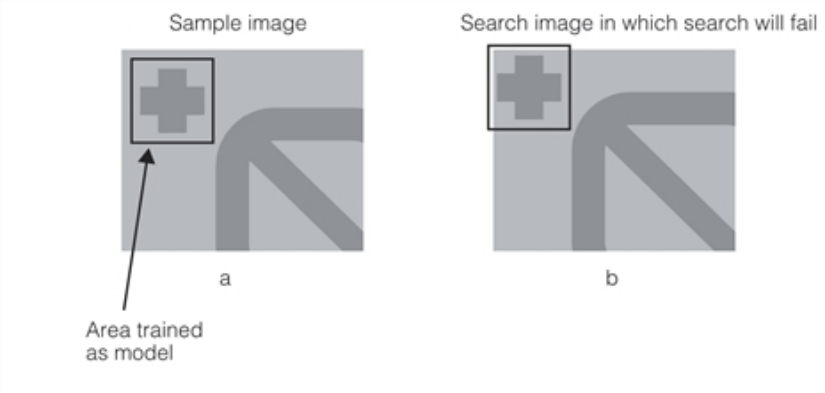
Once you have identified the feature within the model training image that you want to use for a model, you need to define the rectangular region of the image that you actually train as a model. There are two important guidelines to remember when specifying the image area to train as a model.

- CNLSearch only finds instances of the model that are entirely contained within the search image and that are no closer than two pixels to the edge of the search image. CNLSearch does not find model instances that are only partially contained within the search image, and it does not find model instances that extend into a 2-pixel-wide boundary around the outside edge of the image.

  Remember that CNLSearch considers the entire area that you specify to be part of the model.

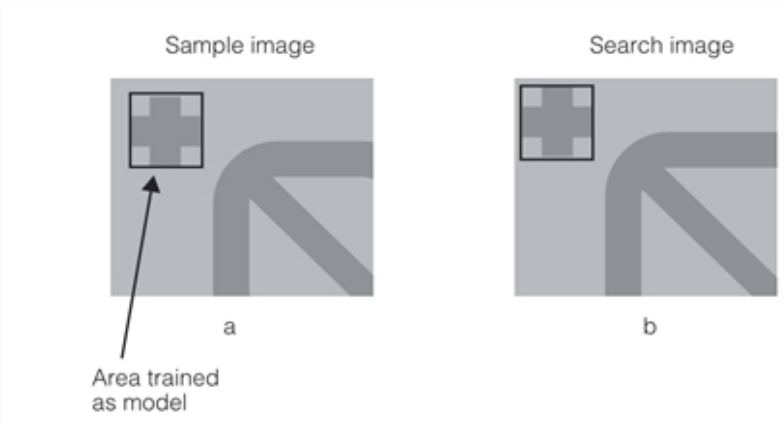
- For nonlinear mode searches, make sure that your model contains as little non-edge-related information as possible. You can display an edge map of the trained model to help identify how much edge information it contains.

The figure below shows an image where an otherwise acceptable model image is trained incorrectly, resulting in search failures. The cross-shaped fiducial in the first sample image (a) is used as the model feature. Because the area used to define the model is so close to the edge of the image, when the search image is shifted (b), the model area falls partially outside the image and CNLSearch fails to locate the model.



Poorly trained model

The figure below shows how the same feature in the same image can be trained correctly. By making sure that sufficient area exists between the edge of the model and the edge of the image, you ensure that the search will succeed even when presented with images in which the position of the feature has moved.

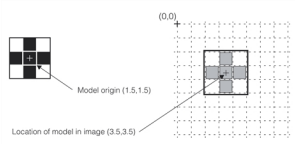


Properly trained model.

Model Origin

When you define and train the model, you can specify its origin. When CNLSearch returns the location where it found an instance of the model within the search image, it returns the point within the search image that corresponds to the origin of the model. If you do not specify an origin for the model, CNLSearch uses the upper left corner of the model as the model's origin.

You can specify the origin of a model to be any point expressed within the model image coordinate system. The origin of a model can be located outside the model. The figure below shows the relationship between the origin of a model and the location of the model in a search image.



Model origin and model location

The origin of a model is used only to determine how CNLSearch returns the location at which it finds the model in the search image. A model's origin has no effect on the speed, accuracy, or reliability of the search.

Minimum Train Region Size

The height and width of any train region you define must meet a minimum size depending on the algorithm you use to perform a CNLSearch inspection:

Algorithm	Minimum Train Region Dimensions
CnlpasLinear	height >= 8 and width >= 8
CnlpasNonlinear	height >= 16 and width >= 16
Search	For training: height x width > 64 For runtime: height x width >= 64 and height >=2 and width >=2 In addition, if your CVL install includes the security license *CVL.CnlSearchSM* you have another option: For training and runtime: height >=4 and width >= 4

Specifying Model Image Edge Threshold

When you define a model to be used for nonlinear mode search, you can specify the edge threshold for the model. CNLSearch uses this edge threshold to construct an edge map from the model image. In most cases, the default edge threshold values work well. You can estimate the effect of different edge thresholds by training a series of models using a single model image with different edge threshold values, then displaying the edge maps produced by different edge threshold settings.

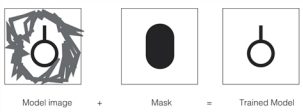
Creating a Scaled Model

Depending on your application, you might need to search images that were acquired at different magnifications from the model image. You can make this task easier by creating one or more scaled versions of your model.

Creating a Masked Model

When you train the model, you can specify a *mask image* in addition to the model image. If you specify a mask image, only those pixels in the model image that correspond to pixels in the mask image with zero values are included in the model.

The figure below shows an example of how you might use a mask image to exclude information from the model image when you train a model.



Using a mask image to train a model

Searching for the Model Within the Search Image

This section discusses in more detail the operations that CNLSearch performs when it searches for an instance of the model image within the search image.

Search Area

When you supply a search image to CNLSearch, it searches the entire area of the image except for a 2-pixel-wide border around the outside edge of the image.

Choosing Between Linear and Nonlinear Mode

As you develop your application using CNLSearch, you need to decide whether to use linear mode search or nonlinear mode search. This section describes how to make that choice.

If your application will encounter only linear brightness changes between search images, linear mode searches will produce accurate results in less time than nonlinear mode searches. In addition, linear mode searches are slightly more tolerant of rotation or scale changes between the model image and the search image, although CNLSearch is not an appropriate choice if you will be experiencing scale and rotation changes.

Because linear mode searches compare pixel values between the model image and the search image, linear mode searches are better at discriminating between valid instances of the model and other features in the image than nonlinear mode searches.

If your application will encounter search images with both linear and nonlinear brightness changes, linear mode works well on the images with linear brightness changes, but poorly on the images with nonlinear brightness changes. Nonlinear mode works equally well on scenes with both kinds of brightness changes, although searches are slower and may not work with even slightly scaled and rotated images.

If you are confident that the search images encountered by your application will undergo linear brightness changes only, you should select linear mode search because of its better performance. If you suspect that you may encounter nonlinear brightness changes, you should select nonlinear mode. A complicating factor in making your decision is that often changes in brightness that appear to be nonlinear are actually linear.

One way to choose between linear mode and nonlinear mode search is to perform a series of test searches on a variety of sample images using both linear and nonlinear mode. If these tests show that linear mode searches tend to fail, produce low scores, or return inaccurate locations more often than nonlinear mode searches, you can assume that the search images have nonlinear brightness changes, and you should select nonlinear mode for your application. If the accuracy of the searches does not vary between linear and nonlinear mode, you can assume that the brightness changes between search images are linear, and you should select linear mode for your application.

You can train your search models for both linear and nonlinear mode searches. Because a model trained for both search modes stores all the information required for both linear and nonlinear mode searches, your application can easily switch between linear and nonlinear mode while it is running. If your application fails to find the model in a particular image or group of images while being used in linear mode, one approach is to temporarily switch to nonlinear mode. Determining whether or not a search succeeds can be very application-dependent. Mismatches in linear mode search can receive high scores.

Keep in mind, however, that the scores returned for a particular search of a particular image are different for linear mode and nonlinear mode. Also, the variability of scores is greater for nonlinear mode searches. If your application will be switching between linear mode and nonlinear mode, you need to determine appropriate acceptance thresholds and confusion thresholds for each mode separately.

Selecting a Confusion Threshold and an Acceptance Threshold

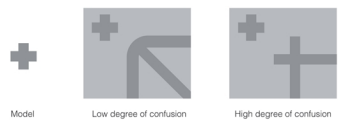
CNLSearch uses the confusion threshold and acceptance threshold that you supply to ensure that the correct instance of the model within the search image is located as quickly as possible. Of the two thresholds, the confusion threshold is the more important to obtaining good results from CNLSearch.

CNLSearch uses both the acceptance threshold and the confusion threshold when considering whether or not a match represents a valid instance of the model. The confusion threshold is the score above which any match is guaranteed to be an instance of the model; all matches with scores greater than or equal to the confusion threshold are considered to be valid. The acceptance threshold is the score at or above which the scores of all valid matches will lie. But other matches, which might not be actual instances of the model, can receive scores above the acceptance threshold.

CNLSearch uses the confusion threshold to speed the search process. If you are searching for a single instance of the model in an image, as soon as CNLSearch finds an instance with a score above the confusion threshold, it stops searching and returns the location of the match. If CNLSearch does not find a match with a score above the confusion threshold, it locates all the matches with scores above the acceptance threshold and returns the location of the match with the highest score.

CNLSearch uses the confusion threshold to determine how to go about discriminating among potential instances of the model within the search image. CNLSearch takes the acceptance threshold as an indication of the degree of image degradation it may expect to encounter.

You should set the confusion threshold high enough to ensure that confusing features in a search image do not receive scores above the confusion threshold. Search images with a high degree of confusion contain features that receive high scores even though they are not valid instances of the model. Search images where the only features that receive high scores are valid instances of the model have a low degree of confusion. The figure below shows examples of scenes with low and high degrees of confusion.



Images with low and high degrees of confusion

Image Confusion in Nonlinear Mode

Because nonlinear mode searches are based on a comparison between edge maps instead of pixel values, some images that do not appear to be confusing to a human observer can be extremely confusing for CNLSearch when it performs a nonlinear mode search.

The figure below illustrates a model image and search image pair that would not be confusing for a linear mode search but that would be confusing for a nonlinear mode search. The upper pair of images in the figure below shows the model image and search image; the search image contains only a single instance of the model, and it appears to have a low degree of confusion. The lower pair of images in the figure below shows the edge maps generated from the model image and the search image. The edge map generated from the search image shows that there are three locations within the search image that have patterns of edges that are very similar to the model image.

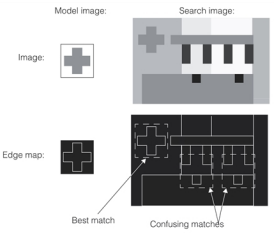
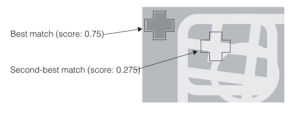


Image confusion in nonlinear mode

Using Test Searches to Select a Confusion Threshold

One technique you can use to estimate the correct confusion threshold is to perform a search with a confusion threshold of 1.0, an acceptance threshold of 0.0, and where you specify that there is one more instance of the model in the search image than there actually is. CNLSearch returns the location and score of both the actual instance and the nearest match. You should select a confusion threshold that lies between these two scores; as a starting point, you might select a confusion threshold that is midway between the scores.

The figure below shows an example of this technique. The search image contains one instance of the model (a cross-shaped fiducial mark). A search for two instances of the model produces a score of 0.75 for the actual instance of the model and a score of 0.275 for the second-best match. Consequently, a good starting value for the confusion threshold in this case would be around 0.5.

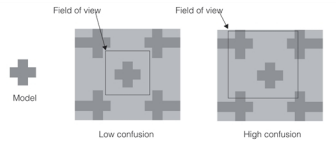


Estimating the confusion threshold

You should perform a series of these test searches, using as many different search images as possible, before selecting a final confusion threshold. Keep in mind that your confusion threshold should be higher than the score received by *any* second-best match.

In many applications, the search image represents only a portion of the overall image area. When you are conducting test searches to select a confusion threshold, you should vary the position of the search image within the overall image area so that you can determine the effect that features that are not normally visible have on the degree of confusion in the image.

The figure below illustrates an image where different fields of view produce different degrees of confusion. You should consider the degree of confusion in parts of the image that are not normally included in the search image when you are selecting a confusion threshold.



Different fields of view from the same image with low and high confusion

Once you have selected a confusion threshold, you can select an acceptance threshold. The acceptance threshold should be less than or equal to the confusion threshold. You should set the acceptance threshold low enough that CNLSearch never rejects an actual instance of the model. As you perform the test searches, note the lowest score that an actual instance of the model receives; you should select an acceptance threshold below this score.

You should keep in mind that a search of the same image for the same model returns a different score for linear mode and nonlinear mode. If your application will be switching between linear mode and nonlinear mode, you should determine the confusion threshold and acceptance threshold independently for the two modes. Also, the scores returned for nonlinear mode searches tend to be more variable than those returned for linear mode searches. In addition, if you adjust the effect of occlusion and clutter on nonlinear mode scores the scores returned for nonlinear mode searches will change.

If you are using nonlinear mode, you should perform enough test searches to be confident about the range of scores that represent valid matches.

Selecting a Search Accuracy

When you perform a search using CNLSearch, you can specify the relative accuracy level for the search. CNLSearch supports *coarse*, *fine*, and *veryfine* searches. The search methods produce increasingly accurate results at the cost of requiring additional memory and time. In addition, the more accurate search methods provide for better discrimination in confusing images. To optimize the performance and space requirements of your application, you should specify the coarsest search method that provides the accuracy and discrimination that your application requires.

The table below lists the accuracy of the different search methods for each of the supported CNLSearch algorithms.

	Accuracy		
Algorithm	Coarse	Fine	Veryfine
CnlpasLinear	± 2 pixels	± 1 pixel	± 0.25 pixel
Search	± 2 pixels	± 1 pixel	± 0.25 pixel
CnlpasNonlinear	± 2 pixels	± 1 pixel	± 0.5 pixel

Search accuracy

The search accuracies listed in the table above represent the best accuracy that CNLSearch can achieve. Depending on the particular image being searched, the actual accuracy may be less than that listed in the table above.

The tables below indicate the relative speed difference for using the different search methods with each of the CNLSearch algorithms.

	Accuracy		
Algorithm	Coarse	Fine	Veryfine
CnlpasLinear	100%	110%	120%
Search	100%	110%	120%
CnlpasNonlinear	100%	150%	200%

Relative search times, score greater than confusion threshold

	Accuracy		
Algorithm	Coarse	Fine	Veryfine
CnlpasLinear	100%	150%	200%
Search	100%	150%	200%
CnlpasNonlinear	100%	200%	300%

Relative search times, score less than confusion threshold

Choosing Between Normalized and Absolute Scoring

For most applications, you should choose normalized scoring. Select absolute scoring only if you want to search for inverted instances of the model. In general, selecting absolute scoring increases the amount of confusion in an image, and it can prevent CNLSearch from finding actual instances of the model.

Specifying a Search Image Edge Threshold

When you perform a nonlinear mode search, you can specify the edge threshold for the search. CNLSearch uses this edge threshold to construct an edge map from the search image. In most cases, the default edge threshold values work well. You can estimate the effect of different edge thresholds by examining the edge maps produced by different edge threshold settings.

Minimum Search Image Size

Depending on the algorithm used, the search image may need to be slightly larger than the training image in order for a search to succeed; using a search image that is too small will cause an exception. For the *eNormalizedCnlpas* and *eNonlinearCnlpas* algorithms, the minimum size for the search image is equal to the size of the training image. For the *eNormalizedSearch* and *eAbsoluteSearch* algorithms, the minimum search image size is typically slightly larger than the training image; an upper bound for the minimum search image size is given by the following formula, rounded to the nearest integer::

$$minX = trainImageWidth \times 1.15 + 0.5$$

$$minY = trainImageHeight \times 1.15 + 0.5$$

where *trainImageWidth* and *trainImageHeight* specify the width and height of the training image.

For CVL users, the **ccCnlSearchModel::minSearchImageSize()** function returns the exact minimum size search image that will work with your trained **ccCnlSearchModel** using the algorithm that you specify; this exact value may be smaller than the upper bound computed by the formula given above.

Search Results

CNLSearch returns a collection of information called a *search result* for each instance of the model that it finds in the search image, up to the number of instances you specify for the search. The results are returned in score order, with the highest score first.

The table below describes the information contained in each search result.

Search Result	Notes
found	A flag indicating whether or not this instance has a score greater than or equal to the acceptance threshold you specified
location	The location at which the instance was found
score	The score the instance received
edgeHit	A flag indicating whether this instance was found at the edge of the search image.
areaScore	The area score for this instance (nonlinear search only)
edgeScore	The edge score for this instance (nonlinear search only)

Search results

Result Statistics

You can obtain a variety of statistical information about the pixels at any location within the search image. You can use this statistical information to perform your own analysis of how the pixel values differ between the model image and the search image. To obtain this information, you supply CNLSearch with an image and a search result that describes the location within the image for which to obtain statistics. *Search results* on page 272 describes the statistical information you can obtain.

Statistical Measure	Description
Sum of Image Pixels	The sum of the values of all of the pixels in the image that correspond to pixels in the model used to generate the search result
Sum of Image Pixels Squared	The sum of the squares of the values of all of the pixels in the image that correspond to pixels in the model used to generate the search result
Sum of Image Pixels times Model Pixels	The sum of the product of each pixel in the search image with the corresponding pixel in the model image
Contrast	The ratio of the standard deviation of the pixel values in the image to the standard deviation of the pixel values in the model image

Statistical information available from CNLSearch

Using Nonlinear Client Coordinates

When the run-time image has a nonlinear client coordinate system, the CNLSearch tool will operate as normal (using the pixels in the image). It will then transform the computed result location (the location of the model origin in the run-time image) through the nonlinear client coordinate system.

Note that since the input image is not transformed before the search, you may see slightly lower scores for model instances found in areas of higher distortion (typically near the corners of the input image). While the scores may be reduced, the use of nonlinear client coordinates will increase the accuracy of the returned model location.

For more information, see the chapter *Using CVL Vision Tools* in the *CVL User's Guide*.

Advanced Topics

The following sections describe some advanced features of CNLSearch. The use of these features is not required for most applications.

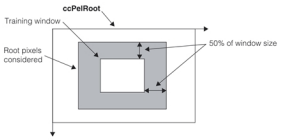
CNLSearch Advanced Training

CNLSearch supports a special advanced training method. You should select the advanced training method only if you experience search failures (such as inaccurate location results) using standard training. The failures that advanced training can correct are usually associated with model training images that contain repeating patterns of evenly spaced features such as grids or sets of parallel lines or bars.

Note: The CNLSearch advanced training method applies only to the *ccCnlSearchDefs::eNormalizedSearch* and *ccCnlSearchDefs::eAbsoluteSearch* algorithms.

If you decide to use the advanced training method, observe the following guidelines:

- Training will take substantially longer using advanced training; as much as several seconds may be required to train a model using advanced training.
- Training will consider pixels outside of the training window in the root image. If possible, supply a training window that is a window onto a larger **ccPelRoot**. The following figure shows the area outside the training window that advanced training may use:



- CNLSearch models trained using advanced training may exhibit differences in search speed than models trained using standard training. Depending on image content, searches might be faster or slower.

Finding Models Partially Outside of the Image

The CNLSearch tool lets you find models that lie partially outside of the search image. This capability is called *partial match searching*.

Restrictions on Partial Match Searching

This section lists the restrictions and limitations associated with partial match searching.

- Partial match searching is only supported for the *ccCnlSearchDefs::eNormalizedSearch* and *ccCnlSearchDefs::eAbsoluteSearch* search algorithms.
- Partial match searching only applies to models that lie partially outside of the search image. Partial match searching is not intended to find partially occluded models within the search image. (Partially occluded models typically receive lower scores or may not be found at all, depending on the extent of the occlusion.)

Training Models for Partial Match Searching

If you enable partial match searching, you should keep in mind that only part of the model you train may be used to match model instances close to the edge of the search image. This can tend to increase the confusion in a search image.

Scoring Partial Matches

CNLSearch lets you specify how you want to score partially matched models. You can compute the score based only on the quality of the portion of the model in the image, or you can further weight that quality score by the fraction of the model that is visible. Weighting the score will produce a lower score.

User-Configurable Overlap Tolerance

The CNLSearch tool lets you specify the amount of tolerance for partially overlapping results. You specify overlap tolerance by specifying the maximum allowed percentage of overlap between two search results. If more than the

specified percentage of the results' areas are overlapped, then the result with the lower score is discarded. If the percentage of overlap is below the value you specify, then both results are returned.

PVE Compatibility

The CNLSearch tool supports compatibility with the PVE software framework in the following ways.

- You can use **ccCnlSearchModel::transmitPveModel()** to save a CVL CNLSearch model as a PVE model. This function writes the CNLSearch model data to the specified file or stream in PVE format using depth bits. The bit depth must be less than or equal to 8 bits.
- You can use **ccPVEReceiver** to read persisted PVE objects from files or streams created by these PVE functions:
  - **ctr_transmit_model()**
  - **cip_transmit_image()**
  - **cnls_transmit_model()**

PatMax

This chapter describes PatMax, Cognex software that lets you locate patterns in images. It contains the following sections:

The first two sections, this one and *Some Useful Definitions on page 275*, provide an overview of the chapter and define some terms that you will encounter as you read.

PatMax Overview on page 275 provides an introduction to PatMax.

How PatMax Works on page 278 describes how PatMax finds patterns in an image.

Using PatMax on page 303 describes the techniques that you use to implement an application using PatMax.

Some Useful Definitions

alignment: The process of running the PatMax tool to search a run-time image for matches on a trained pattern

boundary point: A point along a feature boundary. A boundary point has a location and an orientation (normal to the feature boundary and in the direction of positive intensity change)

clutter: Extraneous features in a run-time image that are not part of the trained model

deformation: A change in a pattern that cannot be described using a linear transformation

deformation rate: A measure of the degree to which a found pattern is deformed from a trained pattern

degree of freedom: Part of a transformation that can be characterized by a single numeric value such as angle

feature: A continuous boundary between regions of dissimilar pixel values. A feature is represented by a list of boundary points

generalized degree of freedom: A degree of freedom other than x-translation or y-translation. For example: uniform scale, x-scale, y-scale, or rotation.

granularity: A measure of the minimum detected feature size in an image.

pattern: A trained (internal) geometric description of an object you wish to find in run-time images.

run-time image: Image in which to locate instances of the trained pattern (also called the target image)

score: Numerical measure of the similarity between the pattern in the trained model and the pattern in the run-time image

shape description: A geometric shape, described from a class derived from **ccShape**, representing the high-contrast boundaries of an object in an image. A shape description represents shape model properties (polarity and weight information) for each boundary it defines. These properties can be implicit (represented by a pure shape) or explicit (represented by a shape model). PatMax can be trained using shape descriptions. Also called a *geometric description*.

transformation: Mathematical representation of the equations that describe the conversion of points from one coordinate system to another coordinate system.

PatMax Overview

This section provides an overview of the key elements of the PatMax software. Like other pattern-location technologies, PatMax trains a pattern, then locates one or more instances of that pattern in one or more run-time images.

PatMax offers three key features that distinguish it from other pattern-location technologies available in machine vision:

- High-speed location of objects whose appearance is rotated, scaled, and/or stretched
- Location technology that is based on object shape, not on grey-scale values
- Very high accuracy

PatMax differs from other pattern-location technologies in that it is not based on pixel grid representations that cannot be efficiently and accurately rotated or scaled. Instead, PatMax uses a feature-based representation that can be transformed quickly and accurately for pattern matching.

Training and Terminology

PatMax can be trained from an image that contains features similar to those you wish to find at run time, or it can be trained from a geometric description of the target features. PatMax training results in a pattern that contains geometric features. See the figure below.

PatMax training

Previous versions of PatMax documentation refer to synthetic training which was associated with the *kSyntheticTrainMethod* used to train PatMax with wireframe objects. The *kSyntheticTrainMethod* training mode is now deprecated and wireframes are now derived from the more general **ccShape** class. We no longer use the term synthetic but rather use the terms *shape training* and *shape description*.

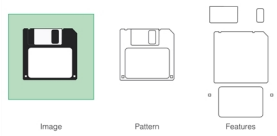
PatQuick and PatMax

The PatMax software supports two pattern-location algorithms: *PatMax* and *PatQuick*. PatMax offers higher accuracy than PatQuick, but requires more time to execute. The PatMax algorithm can also return additional score information.

PatMax Patterns

The images acquired by machine vision systems represent objects as grids of pixel values. PatMax treats images as *patterns*. A PatMax pattern is a collection of geometric features, where each feature is a point on the boundary between two regions of dissimilar pixel values.

The figure below shows an image, the corresponding pattern, and the features that make up the pattern.



Image, pattern, and features

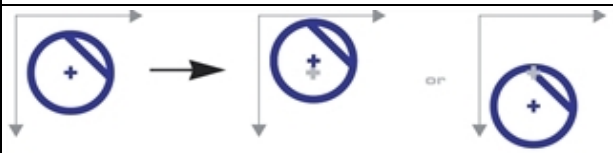
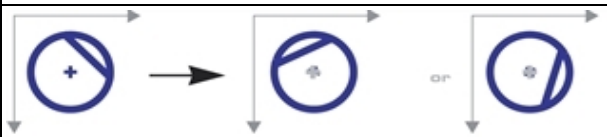
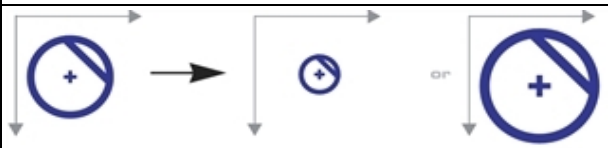
Note: The pattern is made up of both the features *and* the spatial relationship among the features.

Understanding Pattern Transformation

The appearance of an object in an image can vary in several different ways. PatMax can find objects whose appearance varies in any or all of the following ways:

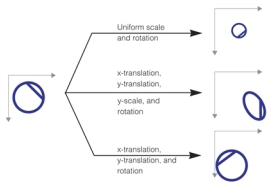
- Size (overall size change or individual x- and y-axis size change)
- Rotation
- Location

Each of these types of transformation is called a *degree of freedom*. Each of these degrees of freedom is illustrated in the table below:

Degree of Freedom	Examples
X translation	
Y translation	
Rotation	
Uniform scale	
X scale	
Y scale	

Degrees of freedom

In many applications, the appearance of an object can undergo several types of transformations at the same time. The figure below shows examples where a pattern experiences transformations in multiple degrees of freedom:



Combining degrees of freedom

What PatMax Does

PatMax finds trained patterns in run-time images no matter what combination of transformations the pattern has undergone. You can limit PatMax to only consider certain degrees of freedom, and within a degree of freedom, only a

specified range. By doing this you can ensure that PatMax finds all of the variations that your application encounters in the smallest amount of time.

For each instance of the pattern that PatMax finds in a run-time image, PatMax returns the location of the instance as well as values for each of the degrees of freedom of the transformation the pattern has undergone.

PatMax also computes a *score* between 0.0 and 1.0 that provides an indication of how closely the pattern in the run-time image matches the trained pattern after accounting for the transformation that the pattern has undergone.

Composite PatMax

The Composite PatMax tool allows you to train a PatMax Composite Model using multiple training instances, which allows more robust pattern matching. This is useful if your run-time images contain noise or clutter or exhibit slight variance. Once you created your Composite Model, it functions like any other PatMax pattern.

Multi-Model PatMax

You can use the Multi-Model PatMax tool if the appearance of the objects you are inspecting varies significantly or you are searching for objects with multiple object appearance types and you want to achieve reliable and simple pattern recognition and alignment with a single tool. A Multi-Model tells you which object appearance type(s) it has found in the run-time image and with what alignment result(s).

An appearance type is defined for the Multi-Model by supplying a trained PatMax model about it to the Multi-Model. Models supplied to a Multi-Model can be standard trained PatMax models or trained PatMax Composite Models, or the mixture of the two.

How PatMax Works

This section provides an overview of how PatMax works. The information in this section will help you understand how to get the most out of PatMax.

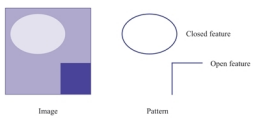
PatMax Patterns

When you train PatMax you specify a region of interest in an image or shape description that includes the features you want to train. When you train using an image, PatMax constructs an internal geometric representation of the features (a pattern) it finds in the training image. These features are made up of continuous boundaries between regions of dissimilar pixels in the image. When you train using a shape description PatMax transforms the model into this same internal geometric description. Once trained, PatMax execution is the same regardless of whether PatMax was trained with an image or with a shape description.

Pattern Features

A PatMax pattern is a collection of features. For shape training, the shape description you provide defines these features. When you train from an image, PatMax isolates all of the features in the image and uses them to create the pattern. An individual feature is defined to be a continuous boundary between regions of dissimilar pixels. The regions can have different intensity, contrast, or texture. A feature can be open or closed.

The figure below shows examples of open and closed features in an image.

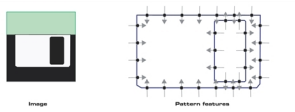


Open and closed features

PatMax pattern features are represented by an ordered list of *feature boundary points*. A feature boundary point has a location and an angle along with links to its neighboring boundary points. The location of a feature boundary point is a point through which the feature boundary passes. The angle of a feature boundary point is the angle between the image

coordinate system x-axis and a line drawn through the feature boundary point perpendicular to the feature boundary and in the dark-to-light direction.

The figure below shows the feature boundary points that define a pair of features.



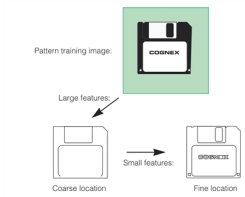
Feature boundary points

Feature Size and Pattern Granularity

The features that make up a pattern can be of different sizes, from features a few pixels in size to features up to 50 or 100 pixels in size. Most patterns contain features with a range of sizes.

PatMax uses pattern features of different sizes to locate similar features in run-time images. In general, PatMax uses large features to find an approximate pattern match in a run-time image quickly, and small features to determine the pattern location precisely.

For example, when PatMax searches for a trained pattern of a diskette, it uses the large features from the diskette (such as the overall shape of the diskette and the outline of the label) to quickly locate the diskette, then it uses the smaller features (such as the letters on the label) to determine the precise location of the pattern. The figure below shows how PatMax uses the different feature sizes.



Large features used for coarse location and small features for fine location

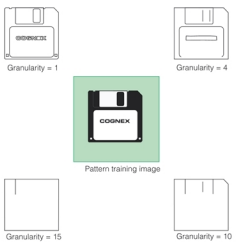
The features that PatMax detects in an image are controlled by the *granularity* used by the PatMax when it analyzes the image. To detect only the large features in an image, PatMax uses a larger granularity setting. To detect the small features in an image, PatMax uses a smaller granularity.

Granularity is expressed as the radius of interest, in pixels, within which features are detected. The figure above illustrates two important characteristics of pattern granularity.

- Large features such as the outline of the diskette are detected at both small and large granularity settings.
- Smaller features are present or absent from the image depending on the granularity setting.

In some cases, however, a feature might be present at a fine granularity and at a coarse granularity, but not at an intermediate granularity.

The figure below shows the effect of different granularity settings on the features that are detected in a single image.



Pattern granularity

At the smallest pattern granularity, the trained pattern includes one or two features for each letter on the diskette label. As the pattern granularity increases, the number of features decreases.

In addition to affecting the features that are trained as part of the pattern, pattern granularity also affects the spacing of boundary points along a feature boundary. In general, the spacing of feature boundary points is approximately equal to the pattern granularity.

PatMax uses a range of pattern granularities when it trains a pattern from an image; PatMax automatically determines the optimum granularity settings when it trains a pattern. The smallest granularity used to detect features in the training image or shape description is called the *fine granularity limit*. The largest granularity used to detect features is called the *coarse granularity limit*.

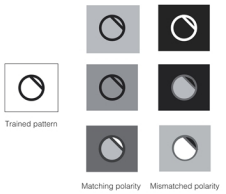
You can obtain a diagnostic display that shows the actual features and feature boundary points trained using the coarse and fine granularity limits. See *Diagnostic Displays* on page 315.

Note: PatMax trains the pattern using a *range* of granularities, not just the coarse and fine granularity limits. The coarse and fine limits are the largest and smallest granularities that PatMax uses.

Pattern Polarity

Each of the boundary points that describes a pattern feature has a *polarity*. The polarity of a boundary point indicates whether the boundary can be characterized as light-to-dark or dark-to-light. You can configure PatMax to find only objects in which every boundary point has the same polarity as the trained pattern, or you can configure PatMax to find objects with mismatched polarity.

Ignoring pattern polarity increases the variety of patterns that PatMax finds. The figure below shows some examples of matched and mismatched pattern polarities. If you configure PatMax to ignore pattern polarity, it finds all of the patterns shown in the figure below. If you configure PatMax to consider pattern polarity, it might not find the patterns in the right-hand column, or it might find them but assign them lower scores than the patterns in the left-hand column.



Pattern polarity

If the run-time images encountered by your application can undergo polarity or other non-uniform changes in brightness, and you want to locate objects that have experienced these kinds of brightness changes, you should configure PatMax to ignore pattern polarity. If you want PatMax to find only patterns with matching polarity, you should configure PatMax to consider pattern polarity.

Note: Ignoring pattern polarity causes PatMax pattern location to be approximately 10% slower than considering pattern polarity.

Pattern Masking

When you train PatMax using an image, you can exclude features from the trained pattern by supplying a mask image. Pattern masking is not supported for shape training since you can design your shape description to include only desired features and thus have no need to mask out unwanted features.

The mask image is interpreted as follows:

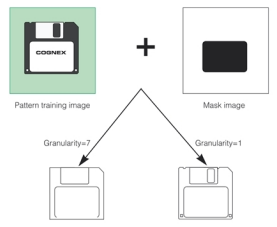
- All pixels in the training image that correspond to pixels in the mask image with values greater than or equal to 192 are considered *care pixels*. All feature boundary points detected within care pixels are included in the trained pattern.

- All pixels in the training image that correspond to pixels in the mask image with values from 0 through 63 are considered *don't care but score pixels*. Feature boundary points detected within don't care but score pixels are not included in the trained pattern.
- When the trained pattern is located in a run-time image, features within the don't care but score part of the trained pattern *are* treated as clutter features.
- All pixels in the training image that correspond to pixels in the mask image with values from 64 through 127 are considered *don't care and don't score* pixels. Feature boundary points detected within don't care and don't score pixels are not included in the trained pattern.

  When the trained pattern is located in a run-time image, features within the don't care and don't score part of the trained pattern are ignored and not treated as clutter features.
- Mask pixel values from 128 through 191 are reserved for future use by Cognex.

Note: If you use the PatQuick algorithm (which does not consider clutter pixels), then mask image pixel values from 0 through 63 are treated the same as mask image pixel values from 64 through 127.

The figure below shows an example of how you would use a mask to exclude features from a diskette label while including features from the diskette in the trained pattern.



Masking image

Using a mask in this case lets you exclude features which might vary between different objects while still using a full range of granularities.

Note: The mask image must have the same dimensions and image offset as the pattern training image.

How PatMax Finds Patterns in an Image

This section describes how PatMax finds and reports pattern matches in run-time images.

Run-time Space

When you search for a PatMax pattern in a run-time image, you define the run-time space that PatMax uses. The run-time space is determined by the degrees of freedom you enable, and the range of values to consider within each degree of freedom.

PatMax identifies likely candidates within the run-time space, then determines the transformation that best describes the transformation from the trained pattern geometry to the transformed pattern geometry in the run-time image.

Pattern Granularity

PatMax locates pattern matches in the run-time space by first searching only for the large features. After locating one or more pattern matches, it uses smaller features to determine the precise transformation between the trained pattern and the pattern match in the run-time image.

PatMax uses the same range of granularities that it computed when it trained the pattern to detect features in the run-time image.

High Sensitivity Mode

Image quality can have a significant affect on PatMax's success in finding features in images. Clear, sharp, high contrast images generally yield the best results while low contrast and noisy images can be problematic. To better enable PatMax to handle this range of image quality, two execution modes are provided. For good images with low noise and high contrast, you should run the tool in *standard mode*, the normal mode of operation. If your images are noisy or have low contrast you may improve your results by running the tool in *high sensitivity mode* which will generally require more training time and more execution time. The figure below shows examples of good and problematic images. Note that high sensitivity mode performance may be worse than standard mode for patterns with small features. For this reason, we recommend standard mode for most applications.

Image examples

Sensitivity Parameter

When you use high sensitivity mode you can also set the *sensitivity* training parameter which allows you to specify your image quality. The sensitivity parameter is a number in the range 1.0 through 10.0 that specifies the amount of noise rejection PatMax applies to run-time images. If the sensitivity parameter is set to 1.0, minimum noise rejection is applied. If the sensitivity parameter is set to 10.0 PatMax applies the maximum noise rejection. Since portions of the pattern may appear as random noise, actual pattern features may be lost. Best results are usually achieved by using the default of 2.0. Lower sensitivity parameter values have less of an affect on training time and execution time whereas higher values tend to increase these times.

If you are having problems when using the default value of 2.0, and you suspect the cause is low contrast or noisy images, you can experiment by changing the sensitivity parameter to determine an optimum setting. Increase the sensitivity parameter to make PatMax less sensitive to random noise, and decrease the sensitivity parameter to make PatMax more sensitive to the pattern. If this helps you should find your optimal setting in the range 1.0 through 5.0. It is very unlikely that a setting above 5.0 will produce an optimal result.

Pattern Weights

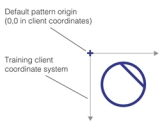
When you train PatMax with a shape description, the shape is generally composed of many primitive shapes encapsulated in a shape tree object that describes the complete shape you wish to find in run-time images. Each of these primitives can be assigned a weighting factor that is used when running the tool to compute a score for how well the trained model matches the likeness found. Primitives with a larger weighting factor have more effect on the score.

Pattern Transformations

When you train a pattern with PatMax either for a shape description or an image, PatMax creates an internal representation of the pattern's geometry. PatMax also initializes the *pattern origin* to a value that you specify. When PatMax returns the location of a pattern instance in a run-time image, it does so in terms of the pattern origin.

Note: In addition to specifying the pattern origin as a simple point, you can also specify a *generalized pattern origin* in the form of a transformation object. The use of a generalized pattern origin is described in the section *Generalized Pattern Origin* on page 284.

By default, the pattern origin is located at the origin of the client coordinate system of the trained model. The figure below shows the default pattern origin for a trained pattern.



Pattern origin

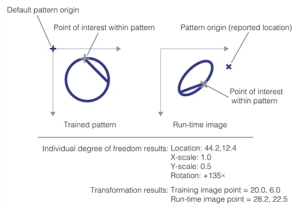
For image training the transform associated with the image transforms image coordinates to client coordinates. For shape training, the shape is defined in client coordinates. However, it is still necessary to specify the client coordinate transformation that maps the shape pattern into training image coordinates (even though there is no training image, per se) so that the approximate resolution at which the pattern should be trained can be determined. Ideally, this resolution should be roughly similar to the resolution of the target patterns in the images on which PatMax will be run.

When you run PatMax, it returns a transformation that describes how the trained pattern maps into the found instance. You can use the information in this transformation in two ways:

- As a transformation object that you can use to convert any location from the trained pattern to the corresponding location in the run-time image

  You can use the transformation object to transform points between the client coordinate system of the trained pattern (translated by any nonzero pattern origin) and the client coordinate system of the run-time image.
- As individual values for the ordinary degrees of freedom (the location of the pattern origin) and individual values for each of the generalized degrees of freedom that you have enabled

The figure below shows an example of a pattern being translated, scaled in the y-axis, and rotated. While you could use the individual values for translation, y-scale, and rotation change to compute where the point of interest is in the run-time image, simply applying the returned transformation object to the point of interest returns the new point of interest directly.



Using the transformation to locate points of interest

If the PatMax programming interface you are using does not support transformation objects, or if you are not using transformation objects in your application, you should set the pattern origin to be a point of interest in the model. If you need to identify the locations of multiple points of interest in the pattern, you will need to perform the appropriate computations using the returned individual degree of freedom results.

The simplest method of performing this computation is to construct a pair of matrices and a vector that describes the individual degree of freedom results, then multiply them by the point of interest.

Note: If you expect any rotation of the pattern in the run-time image, you should place the pattern origin at the center of the pattern, as described in the section *Pattern Origin* on page 309.

The following example shows how to construct these matrices. The example maps a point in the training client coordinate system $P_{(x,y)}$ to the corresponding point in the run-time image client coordinate system $I_{(x,y)}$.

$$\begin{bmatrix} I_x \\ I_y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} P_x \\ P_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$= \begin{bmatrix} s_x \cos\theta & -s_y \sin\theta \\ s_x \sin\theta & s_y \cos\theta \end{bmatrix} \begin{bmatrix} P_x \\ P_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$I_x = P_x s_x \cos\theta - P_y s_y \sin\theta + t_x$$

$$I_y = P_x s_x \sin\theta + P_y s_y \cos\theta + t_y$$

where

s_x and s_y are the x-scale and the y-scale,
θ is the rotation,
t_x and t_y are the x-translation and the y-translation.

Note: The transformation example shown above describes the transformation for a trained pattern with a pattern origin of (0, 0). If you specify a nonzero pattern origin, then the transformation is between a point in the trained pattern client coordinate system *translated by the pattern origin* and the corresponding point in the run-time image client coordinate system.

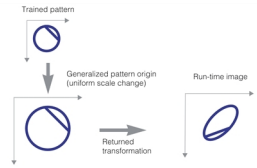
Generalized Pattern Origin

As described in the preceding section, PatMax returns a transformation object that describes how the pattern in the run-time image is different from the trained pattern. The returned transformation object describes the transformation between the trained pattern and the pattern in the run-time image, as shown in the figure below.

Returned transformation (simple origin)

In some cases, you may want to apply a transformation to the trained pattern before you search for the pattern in a run-time image. You apply such a transformation by supplying a generalized pattern origin. PatMax applies the transformation you supply to the trained pattern before it searches for the pattern in the run-time image. When it returns a pattern location result, it returns the transformation between the trained, *transformed*, pattern and the pattern in the run-time image.

The figure below shows how you use a generalized pattern origin to scale a trained pattern before locating it in the run-time image.



Applying a generalized pattern origin transformation

You typically supply a generalized pattern origin to compensate for known scale or rotation changes in the trained pattern. If you supply a generalized pattern origin you should keep the following points in mind:

- PatMax will return results that describe the difference between the trained pattern after it has been transformed by the generalized pattern origin and the pattern in the run-time image.
- Any zone ranges and nominal values that you specify are interpreted with respect to the transformed trained pattern.
- Supplying a generalized pattern origin has no effect on the speed, accuracy, or number of results produced by PatMax.
- The x- and y-translation components of a generalized pattern origin are equivalent to the simple (point) origin.

Score

For each instance of the trained pattern that PatMax finds in the run-time image, it computes a score value between 0.0 and 1.0. The score an instance receives indicates how closely it matches the trained pattern. A score of 1.0 indicates a perfect match; a score of 0.0 indicates that the pattern does not match at all.

When you specify the PatMax algorithm, PatMax scores instances on both the closeness of pattern fit (the degree to which the shape of the features in the run-time image conforms to the shape of the features in the trained pattern) and the presence of clutter (extraneous features). When you specify the PatQuick algorithm, PatMax scores instances on pattern fit only.

In considering the fit, PatMax considers the shape of the pattern. Differences in brightness or contrast (as long as the polarity is the same) are ignored. (You can specify that PatMax ignore polarity changes in addition to brightness and contrast changes.)

PatMax is somewhat tolerant of *elastic stretching* of the pattern. PatMax tends to return lower scores for patterns with missing or extraneous features. The figure below shows examples of patterns with elastic stretching and patterns with missing or extraneous features (called *broken patterns*).

Pattern variations

In all cases, missing features lower the score an object receives. Extraneous features lower an object's score if you specify the PatMax algorithm and if you specify that PatMax consider clutter when computing scores. For more information on scoring and clutter, see the section *Ignoring Clutter When Scoring* on page 291.

Contrast

In addition to the overall score, PatMax also returns the image contrast of each instance of the pattern it finds in a run-time image. The contrast is the average difference in grey-level values for all of the boundary points that PatMax matched between the trained pattern and the pattern instance in the run-time image.

Since PatMax computes the score for a pattern based on the shape of the pattern, the contrast value and score value are generally independent. You can use the contrast value to get additional information about the object.

You can specify a contrast threshold for PatMax searches. If you specify a contrast threshold, only pattern instances where the average difference in grey-level values for all of the boundary points exceeds the contrast threshold are considered by PatMax.

Fit Error, Coverage, and Clutter

When you specify the PatMax algorithm, PatMax returns three additional score values for each pattern instance it finds in the run-time image: the *fit error*, *coverage score*, and *clutter score*.

Note: These scores are only available if you specify the PatMax algorithm.

Fit Error

The fit error is a measure of the variance between the shape of the trained pattern and the shape of the pattern instance found in the run-time image.

The fit error is computed by taking the square root of the sum of the weighted average distances between each boundary point in the pattern and the corresponding boundary point in the pattern instance in the run-time image. If the pattern instance in the run-time image is a perfect fit for the trained pattern, the fit error is 0.0.

You can use the fit error to assess the degree to which the shape of a pattern instance matches the shape of the trained pattern.

Coverage Score

The coverage score is a measure of the extent to which all parts of the trained pattern are also present in the run-time image.

The coverage score is computed by determining the proportion of the trained pattern that is found in the run-time image. If all of the trained pattern is also present in the run-time image, the coverage score is 1.0. Lower coverage scores indicate that less of the pattern is present.

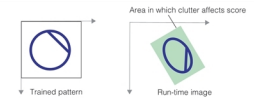
You can use the coverage score to detect missing or occluded features.

Clutter Score

The clutter score is a measure of the extent to which the found object contains features that are not present in the trained pattern.

The clutter score is the proportion of extraneous features present in the found object relative to the number of features in the trained pattern. A clutter score of 0.0 indicates that the found instance contains no extraneous features. A clutter score of 1.0 indicates that for every feature in the trained pattern there is an additional extraneous feature in the found pattern instance. The clutter score can exceed 1.0.

When PatMax computes the clutter score, it considers all features within the area in the run-time image that corresponds to the entire image area used to train the pattern, as shown in the figure below.



Area considered when computing clutter

Controlling PatMax Alignment

This section describes how you control PatMax alignment, the process of finding likenesses of a trained pattern in run-time images.

Degrees of Freedom

When you perform a pattern alignment using PatMax, for each *generalized degree of freedom* (a degree of freedom other than x-translation or y-translation), you must specify either

- That the degree of freedom is disabled, in which case you must specify a nominal value for that degree of freedom. PatMax will only find instances of the pattern that are close to the specified nominal value for that degree of freedom, and PatMax will not compute a value for that degree of freedom; it will report the nominal value that you specify for the degree of freedom.

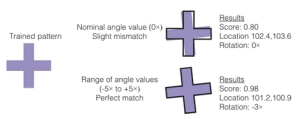
  or

- That the degree of freedom is enabled, in which case you must specify a *zone* that defines the permitted range of values for that degree of freedom. PatMax will find instances of the pattern that have values for the degree of freedom within the specified zone, and PatMax will compute and report a value for the degree of freedom.

If, for example, you are using PatMax to perform translation-only alignment of fiducial marks, you might disable the rotation and scale degrees of freedom and specify nominal values of 0 degrees for rotation and 1.0 for scale.

However, if your application encounters run-time images where the fiducial marks are rotated or scaled very slightly, PatMax will find these instances, but the accuracy of the location information may be slightly reduced and the instance may receive a lower score. PatMax will not compute a value for scale or rotation; it will report the nominal value that you specify for the degree of freedom.

The figure below shows the effect of slight rotation when a nominal rotation value is specified and when a zone for rotation is specified. If a nominal value of 0° is specified for angle, a slightly rotated instance of the pattern is interpreted as a slight mismatch for the pattern. It receives a lower score (0.80 instead of 0.98), a higher fit error, and the accuracy of the position can be reduced.



Effect of setting nominal angle with small angle values

With each additional degree of freedom that you enable, PatMax requires additional processing time to analyze an image. In addition, the larger a zone you specify within a degree of freedom, the more processing time is required. For more information on optimizing alignment speed, see the section *Optimizing PatMax Performance* on page 319.

Finally, PatMax might return some model instances that are slightly outside the zone you specify. For example, if you specify a scale between 0.95 and 1.05, PatMax might return results with scale values of 1.09 or 0.91. You can always check the result values and exclude results that are outside the specified zone.

Specifying a nominal value for a degree of freedom lets you find patterns with small variations in the degree of freedom at the highest possible speed, since the degree of freedom is not part of the computation.

Image Confusion

As you increase the number of degrees of freedom and as you enlarge the zone for an individual degree of freedom, you increase the number of potential matches in a run-time image.

The figure below shows one effect of enabling a degree of freedom on the confusion of the image. With the scale degree of freedom disabled, only a single instance is found. With scale enabled, the top of the pattern is a fair match for a scaled instance of the entire pattern.

Image confusion increased with larger run-time space

In general, the larger the run-time space, both in terms of the size of the zone and in terms of the number of enabled degrees of freedom, the greater the potential for image confusion.

Effect of Non-Uniform Scale Degrees of Freedom

Enabling the non-uniform scale degrees of freedom in PatMax reduces the accuracy level to that of the PatQuick algorithm.

The x- and y-scale degrees of freedom allow you to specify variations from training to run-time objects that are called non-uniform scale variations, and provide alternative ways of specifying variations in aspect ratio from trained patterns to runtime objects. As with all generalized degrees of freedom, non-uniform scale variation can be specified using either nominal or zone parameters. Whenever a non-uniform scale zone is enabled, PatMax accuracy is limited to the lower accuracy of the PatQuick algorithm. Using non-uniform scale nominal values other than the default 1.0 results in no loss of accuracy.

If the aspect ratio of your objects truly varies, you may have no choice but to use a non-uniform scale zone and accept the lower accuracy. In some applications, however, aspect ratio does not vary from object to object, but is nevertheless different from the trained pattern. Do not use non-uniform scale zones to get PatMax to find the correct aspect ratio, because accuracy will be limited to the lower accuracy of the PatQuick algorithm. Instead use non-uniform scale nominal values, client coordinates, or pattern coordinates to specify the fixed variation in aspect ratio from training to run-time.

Expected Result Count and Score Thresholds

When you search for a pattern using PatMax, you specify the number of instances you expect to be present in the run-time image and a score threshold that specifies the minimum score value that an actual instance of the object is expected to receive.

PatMax returns information about all of the objects in the run-time image that receive scores above the threshold you specify up to the number of instances that you specify.

If there are multiple instances with similar scores, PatMax may return information about more instances than you specify. For example, if you specify a score threshold of 0.50 and an expected result count of 2, and PatMax finds five instances with the following scores:

```
0.98
0.70
0.68
```

```
0.32
0.19
```

PatMax will return information about the first three instances, even though you only requested two. PatMax does this because of the potential ambiguity between the two results with scores of 0.70 and 0.68.

In addition to using the score threshold to determine the number of results to return, PatMax also uses it to help refine the alignment process. If you specify a low score threshold, PatMax might take somewhat longer to execute because it must consider more potential matches for the trained pattern. If you specify a high score threshold, PatMax can quickly eliminate potential matches.

Note: In some cases, PatMax may not find pattern instances with scores slightly above the score threshold you specify.

Overriding Thresholds during Coarse Search

As described in the section *Feature Size and Pattern Granularity* on page 279, PatMax performs an initial coarse search using a large granularity value, then refines the search using a smaller granularity value. To avoid discarding potential pattern instances during the coarse search, PatMax uses a fraction of the specified accept threshold to decide whether to continue refining a given instance. Under some circumstances, this coarse threshold fraction can be too aggressive, causing PatMax to discard results that would, after refinement, receive scores above the specified accept threshold.

You can override the coarse accept threshold fraction. To help you set this value appropriately, PatMax results include the maximum coarse accept threshold value that the instance could have received.

Non-Linear Pattern Deformation

By default, PatMax requires that each boundary point in the instance of a pattern found in a run-time image closely correspond to a boundary point in the trained pattern. PatMax can match and identify any change that can be described by a linear geometric transformation (assuming you specify the appropriate degrees of freedom and zones).

When patterns exhibit nonlinear geometric changes, PatMax can fail to find them, or it can return a low score or inaccurate location information. There are two ways PatMax can work with patterns that exhibit nonlinear geometric change.

- In cases with a small amount of deformation, you can specify a nonzero *elasticity value* (in pixels) that specifies how much non-linear deformation PatMax will tolerate.
- In cases with substantial deformation, you can use the *PatFlex* algorithm to locate and match patterns and return information about the deformation of the found pattern instance. In cases with perspective distortion only, you can use the parameter-optimized *PatPersp* algorithm as well.

Each of these two methods is described in this section.

Elasticity

You can specify the degree to which you will allow PatMax to tolerate nonlinear deformation by specifying an *elasticity value*. You specify the elasticity value in pixels.

In general, you should specify a nonzero elasticity value if you expect inconsistent variation in patterns in run-time images.

You should keep the following points in mind when specifying a nonzero elasticity value:

- Specifying a nonzero elasticity value does not affect PatMax's execution speed.
- Increasing the elasticity value does not decrease PatMax's accuracy. However, location information returned about additional object instances that are found as a result of increasing the elasticity value can be less accurate.
- If the elasticity value is too low, you will see low scores and your application may fail to find patterns in the run-time image and/or the positions will be incorrect or unstable.
- If the elasticity value is too high, PatMax may match false instances and may return inaccurate or unstable results.

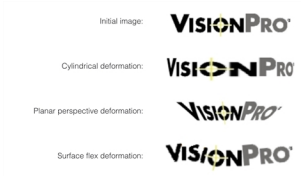
In general, you should start with an elasticity value of 0; if necessary, increase the value slowly until you obtain satisfactory results.

PatFlex Algorithm

If you specify the PatFlex algorithm, PatMax can tolerate and report information about a high degree of nonlinear deformation. Some typical types of pattern deformation that PatFlex is appropriate for are listed below:

- Planar perspective distortion
- Spherical or cylindrical distortion
- Surface flex distortion

Each of these types of deformation is shown in the figure below.



Deformation types

Understanding PatFlex Parameters

If you use the PatFlex algorithm, you have the opportunity to specify several parameters.

- The estimated deformation rate (training time), and maximum deformation rate (run-time)
- The deformation transformation type
- The number of control points
- The smoothing value
- The refinement mode
- The partial match mode

Each of these parameters is discussed in this section.

Maximum and Expected Deformation Rate

Different instances of a trained pattern can exhibit different amounts of deformation. The PatFlex algorithm expresses the degree of pattern deformation as the pattern's *deformation rate*. A pattern's deformation rate is a value in the range 0.0 through 1.0. A value of 0.0 indicates no deformation, while a value of 1.0 indicates a highly deformed pattern.

The deformation rate represents the percentage of variance between the undeformed location of a control point in the run-time image and the found location of the control point. The figure below provides a visual guide for estimating pattern deformation.

Pattern deformation rate

You specify two deformation rate values: an estimated deformation rate used with the trained pattern, and a maximum deformation rate for a particular input image (specified at run time).

Deformation Transformation Type

If you expect the deformation in your run-time image to be limited to planar perspective distortion, you can limit PatFlex to detecting that type of deformation. Limiting the deformation type in this way greatly increases accuracy.

Control Points

You can specify the number of control points that PatFlex uses to map the pattern. By default, PatFlex creates a grid of control points evenly spaced across your pattern. In most cases this produces a satisfactory balance of reasonable execution time and accurate representation of the pattern's deformation.

If you expect high-frequency deformation, such as a crumpled piece of paper, you may need to increase the number of control points.

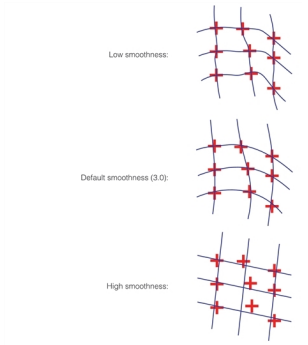
You can also specify the location of the control points explicitly. You should consider specifying points explicitly if you expect differing amounts of deformation at specific locations within the pattern.

Smoothness Value

PatFlex creates a nonlinear transformation that describes the difference between the trained control points and the control points found in the run-time image. PatFlex refines this transformation so that you can use it to map any point between the trained pattern and the run-time image or create an unwarped version of the run-time image.

The *smoothness* value determines how closely PatFlex fits this transformation to the found control point locations. A smoothness value of 0.0 indicates that the transformation is to be perfectly fitted to the control points. A smoothness value of infinity indicates that the transform is an affine transform (does not incorporate any of the nonlinear deformation).

In most cases, the default value of 3.0 is appropriate. If too small a smoothness value is specified, the resulting transformation may be perfectly accurate for the control points, but less accurate for intervening points, as shown in the figure below.



Effect of different smoothness values

Refinement Mode

In addition to specifying a smoothness value, you can also specify the *refinement mode*. The refinement mode determines how finely PatFlex refines the computed transformation. The table below defines the available refinement mode values.

Refinement Mode	Description
None.	No attempt is made to remove residual error from the computed transformation.
Coarse	The computed transformation is refined so that it has no more residual error than the specified coarse granularity limit.
Medium	The same as coarse refinement, except that certain types of high-level inaccuracy are removed.
Fine	Same as medium refinement, except that the transformation is refined so that it has no more residual error than the specified fine granularity limit.

PatFlex refinement modes

Partial Match Mode

You can specify the *partial match mode*. If partial match mode is enabled, PatFlex accepts any candidate match for which it can match a proportion at least equal to the specified coverage threshold with a score on those sections of the pattern greater than or equal to the specified accept threshold.

PatFlex Performance

The PatFlex algorithm generally requires substantially more time for both training and searching, than the other PatMax algorithms. Also, as you increase the number and size of the enabled degrees of freedom, PatFlex may require significant amounts of time to run. Finally, specifying large numbers of control points will increase search times. The expected deformation rate (specified at run time) has a similar effect on performance; the larger the expected deformation rate, the longer the search may take.

PatFlex Results

In addition to returning a transformation that describes the pattern deformation, the PatFlex algorithm also returns all of the standard PatMax results, such as the pattern score and a linear transformation describing the result's pose. The returned pose represents a best-fit linear approximation.

PatPersp Algorithm

In case the only nonlinear distortion your run-time images will exhibit is planar perspective distortion, you can use the Perspective PatMax (PatPersp) algorithm for the best speed performance. PatPersp differs from PatFlex in that it performs pattern matching using optimized and fixed PatFlex-only train-time and run-time parameters. These fixed PatFlex-only train-time and run-time parameters are optimized for speed while maintaining approximately the same accuracy level as PatFlex has using its planar perspective distortion option. (In most cases PatPersp accuracy is better than PatFlex accuracy.) This means that any PatFlex-only train-time or run-time parameters you specified are ignored while using PatPersp.

PatPersp Performance

PatPersp provides significant speed performance increase in most cases compared to using PatFlex with the planar perspective distortion option.

Ignoring Clutter When Scoring

When you specify the PatMax algorithm, you can tell PatMax to ignore clutter when computing the score of an instance of the pattern in the run-time image. If you tell PatMax to ignore clutter, then pattern instances receive the same score regardless of the presence of extraneous features.

The figure below shows the effect of ignoring clutter when scoring instances.

Computing score with and without clutter

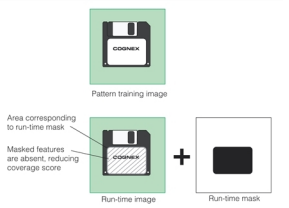
Note: If you specify the PatQuick algorithm, PatMax *always* ignores clutter when scoring.

Run-Time Image Masking

In addition to specifying a mask for a pattern training image, as described in the section *Pattern Masking* on page 280, you can specify a second mask for the run-time image. Any mask image you specify must have the same dimensions and image offset as the run-time image.

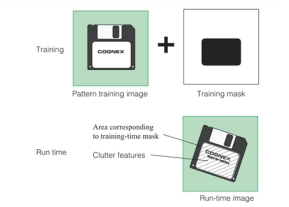
PatMax will exclude from the run-time image any features that lie within *don't care pixels*. Don't care pixels are defined as all pixels with values less than 128. Pixels with values greater than or equal to 128 are considered *care* pixels. Features that lie within care pixels are retained.

Features that are present in the trained pattern but masked from the run-time image will reduce the coverage score received by found pattern instances, as shown in the figure below.



Example of run-time masking

Features in the run-time image that correspond to don't care but score portions of the trained pattern will be scored as clutter features, as shown in the figure below. This is because those features are not part of the trained pattern.

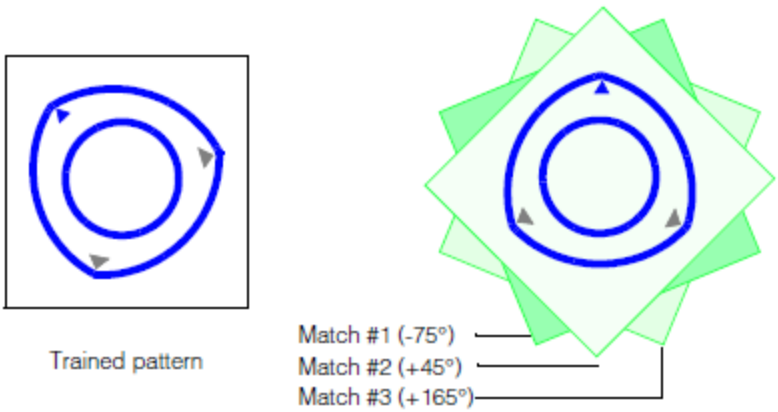


Clutter features and masking

You can prevent PatMax from treating those features as clutter features by using the don't care and don't score mask value when you train the pattern. For more information, see the section *Pattern Masking* on page 280.

Controlling Overlap Tolerance

By default, when PatMax finds multiple pattern instances in the run-time image that largely overlap each other, PatMax assumes that these instances actually represent the same pattern in the image. For example, if you use PatMax to search for the part shown in the figure below with the angle degree of freedom enabled, PatMax will locate three instances of the part at the same location but at three different angles. By default, PatMax discards all but the strongest match and returns a single result.



PatMax selects the best of overlapped instances

For some applications, it might be necessary or desirable to obtain a separate result for each of the matching instances shown in the figure above.

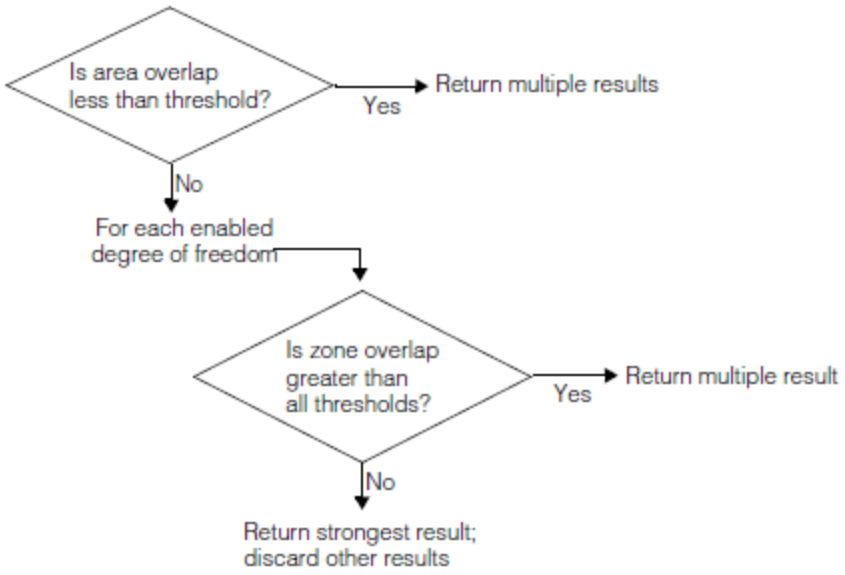
PatMax lets you control how multiple overlapping pattern instances are handled by letting you specify an *overlap threshold*. You can specify an area overlap threshold (expressed as the percentage of the pattern training area that overlaps) and a separate zone overlap threshold for each enabled degree of freedom.

Note:
The area overlap and zone overlap thresholds behave differently.

As you increase the area overlap threshold, PatMax requires that multiple instances overlap to a greater degree before it treats them as a single instance. As you decrease the area overlap threshold, PatMax requires that multiple instances overlap to a lesser degree before it treats them as a single instance.

As you increase a zone overlap threshold, PatMax requires a smaller distance between multiple instances before it treats them as a single instance. As you decrease a zone overlap threshold, PatMax requires a greater distance between multiple instances before it treats them as a single instance.

PatMax uses the procedure shown in the figure below to determine whether or not to discard overlapped pattern instances.



Overlap threshold processing

Degenerate Results

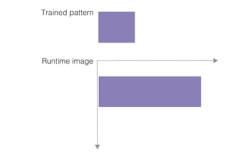
For a given trained pattern, set of enabled degrees of freedom, and run-time image, PatMax returns the transformation that best describes the appearance of the trained pattern in the run-time image. For some combinations of patterns, degrees of freedom, and run-time images there are multiple equally correct transformations.

For example, if you train a circular pattern, then search for that pattern with the angle degree of freedom enabled in an image containing a circle, there are an infinite number of equally valid transformations. The combination of such a trained pattern, enabled degrees of freedom, and run-time image is called a *degenerate system*. Each of the results is called a *degenerate result*.

If a particular PatMax result is degenerate, that means that there are other equally accurate transformations that describe the difference between the trained pattern and the object in the run-time image. You can determine whether or not a particular result is degenerate by requesting diagnostic information for that result. See *Run-time Information Strings* on page 315.

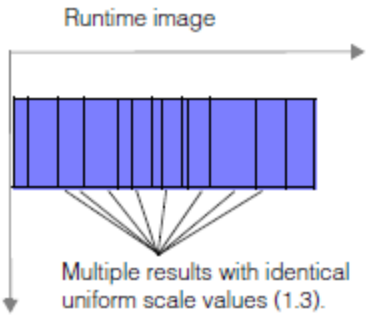
Note: In most cases, PatMax will have detected the potential for degenerate solutions when you trained the pattern.

The figure below shows an example of a trained pattern and run-time image that produces multiple degenerate results when the uniform scale degree of freedom is enabled.



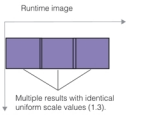
Pattern and run-time image with potentially degenerate solutions

Any attempt to locate the trained pattern shown in the figure above in the run-time image shown in the figure above with the uniform scale degree of freedom enabled will result in an infinite number of identically transformed instances of the trained pattern in the run-time image. Each instance has the same degree of uniform scale change, and each is located at a different offset within the horizontal line in the run-time image. The figure below shows some of these degenerate results.



Degenerate solutions

The actual number of degenerate results that PatMax returns in a situation like the one shown in the figure above is determined by the overlap thresholds that you configure for PatMax. For example, if you specified an area overlap value of 5%, then PatMax would return approximately the number of results shown in the figure below.



Number of degenerate results limited by overlap threshold

Because the additional pattern instances shown in the figure above (entitled Degenerate solutions) overlap each other by more than 5%, they are discarded by PatMax.

Individual regions of a PatFlex pattern (or PatPersp pattern) might be locally degenerate in which case the resulting transform may not have correct information about some aspects of the pattern. For example, if part of the pattern is a circle with no other features around it, the region containing only the circle would be degenerate in that the transform has no valid information about rotational changes.

Working with Non-square Pixels

If your camera has non-square pixels and your application requires high-accuracy location of rotated patterns, you must calibrate your system and apply the calibration to run-time images and to training images when training from an image. When using shape descriptions, the shapes should be specified in the client coordinate system defined by the calibration, and that the calibration transform should be used as the client coordinate transformation when training.

Composite PatMax

Composite PatMax allows you to create PatMax patterns from multiple training images. Composite training limits the features added to the pattern to those that appear in most (or all) of the training images, and it excludes from the pattern those features that only appear in a few of the image.

Composite PatMax allows you to solve several types of applications that can be difficult when you are limited to pattern training from a single image.

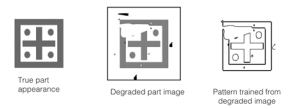
- Applications where no representative image of your part exists.
- Applications where you need to ignore certain features that vary from part to part.
- Applications where a part appears in different backgrounds.

Each of these application types is described in this section.

Applications without a Representative Image

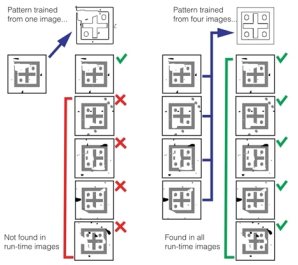
Conventional PatMax works by training a pattern based on the features found in a representative (or “good”) image of your part. In some applications, it may be impossible to acquire a part image that is not affected by noise, clutter, occlusion, or other defects.

Attempting to train a conventional PatMax pattern from such a degraded image often produces an unusable pattern, since the pattern includes numerous features that are not present in other run-time part images, as shown in the figure below.



Pattern trained from unrepresentative image

You can train a nearly ideal Composite Model using multiple degraded training images. Composite Model PatMax collects the common features from each image and unites them into a single ideal model. This way, you can filter out noise or other random errors from the training images that would appear in the final Composite Model, as shown in the figure below.



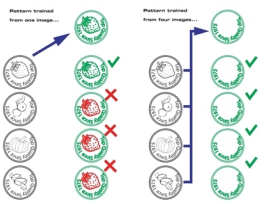
Composite model training from multiple degraded images

As shown in the figure above, the composite-trained pattern successfully finds the part in both the degraded images used for training, and in newly encountered images.

Parts With Variable Features

In some applications, you need to be able to locate and align parts in which certain features may be different in different parts. For example, in the case of a product packaging line, packaging lids may have some features in common (the product name) while other features are different from part to part (the product flavor).

The figure below shows how you can use composite PatMax training to create a pattern that includes the features that are the same for all parts while excluding the features that are different from part to part.

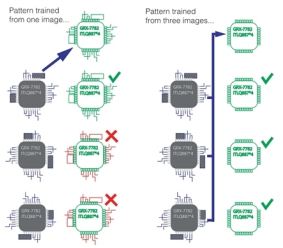


Composite PatMax training for parts with varying features

Training with Varying Background

Composite Modeling can also be used to filter out background changes. For example, it allows you to train a pattern that you can use to locate a component at different locations on a printed circuit board, where the background, consisting of electrical connection (tracks and solder) differs at each location.

Composite PatMax training is particularly useful when you need to use images of the part already attached to the circuit board for training, as shown in the figure below.



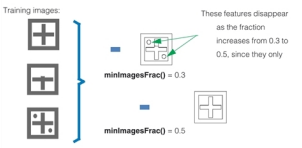
PatMax Composite for parts with different backgrounds

Controlling Composite Training: Minimum Images Fraction

For a model feature to be included in the Composite Model, it must be present in at least the minimum images fraction (**ccPMCompositeModelParams::minImagesFrac()**) of the images.

For example, if you have 4 training images and the minimum images fraction is set to 0.5 (50%), then a feature will have to be present in at least 2 images to be included in the Composite Model; if you have 5 training images and the minimum images fraction is set to 0.5 (50%), then a feature will have to be present in 3 (60% of the) images; 2 would be 0.2 (40%) and therefore below the threshold.

PatMax Composite for parts with different backgrounds on page 296 shows the effect of the **minImagesFrac()** parameter.



Using the **minImagesFrac()** parameter

Using Composite Model PatMax Training

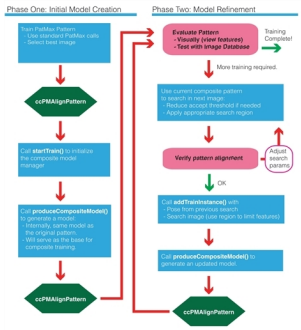
This section provides an overview of how you use PatMax composite model training. For additional information, see the header file *ch_cvl/pmcmpmod.h.*

Using the PatMax Composite Model Manager

PatMax composite model training starts with a standard PatMax pattern that you train using the “best” part image that you can obtain. You then use the **ccPMCompositeModelManager** object to manage, create, and train Composite Models for PatMax, starting with your standard PatMax pattern.

The figure below provides an overview of how you use the Composite Model Manager.

Note: The function names in the figure below are member functions of the **ccPMCompositeModelManager** class.



Recommended workflow with Composite Model PatMax

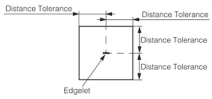
You begin Composite Model creation by calling **ccPMCompositeModelManager::startTrain()** supplying an initial trained single PatMax model to it. This model is used as the first train instance, and many of its settings are carried over to the Composite Model. This includes the granularities, which algorithms are trained, and the definition of the model coordinate space. Next, you add fixtured model instances to the **ccPMCompositeModelManager** object using **ccPMCompositeModelManager::addTrainInstance()** (or **ccPMCompositeModelManager::addTrainInstances()** if you are adding multiple model instances). You supply to **ccPMCompositeModelManager::addTrainInstance()** an input training image paired with a *modelFromClient* "pose" transform that tells where the model is found in that image. At any time after **ccPMCompositeModelManager::startTrain()** has been called, the manager can produce a Composite Model based on all added train instances by a call to **ccPMCompositeModelManager::produceCompositeModel()**.

You may repeat this process as often as desired, adding a new instance and requesting a new Composite Model.

There are a number of parameters to guide the process of creating a Composite Model, which are specified in the **ccPMCompositeModelParams** object.

Correspondence Distance Tolerance

During composite training, the distance tolerance (**ccPMCompositeModelParams::matcherDistanceTolerancePels()**) is used to limit how far away a matching feature can be. Each image's features are corresponded to each other image's features. For example, with the edgelet shown below, the capture range would look as follows:



This parameter can be used mainly for speed, reducing the number of correspondences to consider. The internal feature matcher will always prefer closer correspondences. It is far more convenient to specify this value in pels, so that a standard default can be meaningful for many applications. Pel units from the most recently added training instances are used.

Note: Composite pattern training may generate synthetic pattern features that do not exactly correspond to the features in any specific training image.

This tool is not intended for use when the images contain fundamentally different models. In this case, the behavior is still as outlined, but this may produce a subtly unusable model. For example, you could create a single model that is the composite of three unique models, by setting the **ccPMCompositeModelParams::minImagesFrac()** to approximately 0.25. However, to actually find instances of these models at run-time, the PatMax accept threshold would also have to be around 0.25, which likely would make it impossible to avoid spurious matches. When there are multiple distinct models, Cognex recommends that you use the PatMax Multi-Model interface, which can be used in combination with Composite Models.

Ignoring Polarity

If your train-time (and likely run-time) instances may exhibit polarity reversal, or any kind of polarity variance, you can choose for Composite PatMax to ignore polarity by setting **ccPMCompositeModelParams::ignorePolarity()**. (See

Pattern Polarity on page 280 for a detailed description of polarity.) In this case, the created Composite Model will have its **ccPMAlignPattern::ignorePolarity()** feature set to true. This is because a composite PatMax model produced with the **ccPMCompositeModelParams::ignorePolarity()** feature set to true cannot be usefully run with the **ccPMAlignPattern::ignorePolarity()** feature set to false because the original direction information is lost.

If **ccPMCompositeModelParams::ignorePolarity()** is false, then the Composite Model will have its **ccPMAlignPattern::ignorePolarity()** set to match that of the initial **ccPMAlignPattern** supplied to **ccPMCompositeModelManager::startTrain()**.

Training Image Order

The order of images supplied during composite training may affect the quality of the resulting composite model. In general, composite training works best when images that exhibit slow and continuous appearance changes between images produce the best results.

Retraining Using Different Granularity Limits

You can retrain your existing Composite Model using different granularity limits by calling **ccPMCompositeModelManager::retrainGrainLimits()** with the new granularity limits, which retrains the internal data to use the new specified granularity limits, and then calling **ccPMCompositeModelManager::produceCompositeModel()**, which will produce a new Composite Model trained with these new granularity limits.

To be able to do this, it is required that the original training images and their pose transforms are saved during their additions by setting **ccPMCompositeModelManager::saveImages()** to true.

Optimizing Composite PatMax Performance

Composite Model training typically uses significantly more memory and execution time than training a single PatMax model.

You can limit the memory usage of Composite Model generation by setting the **ccPMCompositeModelManager::maxNumTrainInstances()**. If additional instances are added to the model beyond this number, the resulting behavior is defined by **ccPMCompositeModelDefs::trainInstanceOverflowHandling**:

- Throwing:
  The **ccPMCompositeModelManager::addTrainInstance()** or **ccPMCompositeModelManager::addTrainInstances()** function throws *ccPMAlignDefs::CanNotTrain* if the total number of train instances would become greater than **ccPMCompositeModelManager::maxNumTrainInstances()**.
- Ignoring:
  Train instances beyond **ccPMCompositeModelManager::maxNumTrainInstances()** are silently ignored.
- FIFO behavior:
  The least-recently-added train instance(s) are discarded in favor of the most-recently-added train instance(s) to maintain a total of no more than **ccPMCompositeModelManager::maxNumTrainInstances()**.
  For eFIFO to operate as intended, **ccPMCompositeModelManager::saveImages()** must be true.

For additional speed, you can set **ccPMCompositeModelParams::matcherDistanceTolerancePels()** as described previously.

Multi-Model PatMax

If the objects you are inspecting have multiple discrete appearance types and the appearances vary significantly and you want to achieve reliable and simple pattern recognition and alignment with a single tool, you can use a PatMax Multi-Model (referred to as Multi-Model hereinafter). That is, the appearance of the objects may fall into clear distinct categories (unlike in the case of the Composite Model where typically the appearance is expected to vary continuously). A Multi-Model tells you which object appearance type(s) it has found in the run-time image and with what alignment

result(s). An alignment result includes, for example, the location, pose transform, and score (which is a number between 0.0 and 1.0) for each found instance.

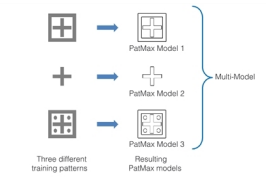
The Multi-Model contains the trained PatMax models you add to it, and it performs pattern alignment and classification on your run-time images based on the added trained PatMax models and the run-time parameters you specify. This means that if you run a Multi-Model on a supplied run-time image containing an object instance, it returns the alignment result from the PatMax model that produces the best result.

The main advantage of using a PatMax Multi-Model over using individual PatMax models is that at run-time, the Multi-Model carries out feature extraction once per run-time image (which features are then input to each model) as opposed to running individual PatMax models which would cause identical feature extraction to occur in every model. See more in *Advantage of Using PatMax Multi-Model over Using Individual Models – Pattern Granularity with Multi-Model* on page 300.

You can supply either standard PatMax models or PatMax Composite Models to the Multi-Model, or the mixture of the two.

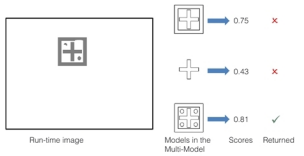
Using Multi-Model to Discriminate Between Multiple Different Patterns

You can use a Multi-Model to discriminate between multiple different patterns. The following figure illustrates how a Multi-Model is created with standard PatMax models.



Multi-Model creation in combination with Composite Model training

The following figure illustrates pattern classification based on the previous models added to the Multi-Model.



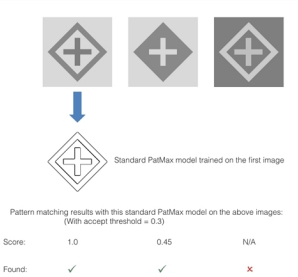
Multi-Model pattern matching with added Composite Models

In this case, the third model from above received the highest score (0.81); therefore, the pattern alignment result is returned for this model.

Using Multi-Model to Discriminate Between Varying Appearances of the Same Object

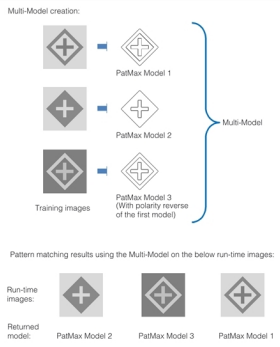
When the appearance of your object under inspection varies, for example, being in different production stages, you can use Multi-Model to find your object and determine in which stage it is in the current run-time image.

The following figure shows an object in three different production stages and the run-time results of running the standard PatMax model trained on the first image.



Standard PatMax model pattern matching with an object whose appearance varies

The following figure shows how a Multi-Model is created with the previous images and how it is used to discriminate between the images.



Multi-Model creation and pattern matching with an object whose appearance varies

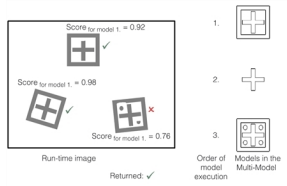
Advantage of Using PatMax Multi-Model over Using Individual Models – Pattern Granularity with Multi-Model

One of the key advantages of using the PatMax Multi-Model over using individual PatMax standard or Composite Models is efficiency. At run-time, the Multi-Model carries out feature extraction once per run-time image. These features are then used as input to each of the models to be run. (This is as opposed to running individual PatMax models which would cause identical feature extraction to occur in every model). To enable this efficiency, a restriction is imposed on all models added to the Multi-Model: they must all have identical coarse granularities and identical fine granularities. The Multi-Model provides an API to automatically retrain its component PatMax models to ensure their granularities match and thus meet the above condition.

Model Search Modes – Using the Confusion Threshold Instead of Exhaustive Search

You can configure the Multi-Model so that a confusion threshold that you define is used as the criterion for returning a model instead of running all models and selecting the one with the highest score. In this case, models are run in a predefined order (queue) until enough results are found with scores greater than or equal to the confusion threshold. If Multi-Model finds no models scoring greater than or equal to the confusion threshold, then the single model results from the highest scoring model are returned.

The following figure illustrates using a confusion threshold of 0.9 and setting the number of models to find to 2.



Multi-Model pattern matching using a confusion threshold

The two models scoring greater than the confusion threshold and their results are returned.

Multi-Model PatMax Usage Details

A PatMax Multi-Model, of type **ccPMMultiModel**, is a collection of PatMax models, either standard models of type **ccPMAlignPattern** or Composite Models stored as Composite Model Managers of type **ccPMCompositeModelManager**. A **ccPMMultiModel** tells you which PatMax models it has found in the run-time image and with what alignment results. Multi-Model results are returned as a **ccPMMultiModelResultSet** object.

Operation

First, make sure your standard PatMax models and/or Composite Models to be added to the Multi-Model are available.

Then, add your models to the Multi-Model one by one. You can add a standard PatMax model using the **ccPMMultiModel::addModel()** function and a Composite Model using the **ccPMMultiModel::addCompositeModel()** function. These functions return an internally assigned ID for the added model. You can access an added model by calling the **ccPMMultiModel::model()** function by supplying the model ID. If you want to remove a model, you can call the **ccPMMultiModel::removeModel()** function by supplying the model ID.If necessary, some or all of the added models are retrained during addition to ensure that their granularities match as defined by the supplied *grainLimitPolicy* parameter. The valid values of this parameter are defined in **ccPMMultiModelDefs::GrainLimitPolicy**.

A Multi-Model makes a copy of each added **ccPMCompositeModelManager** and **ccPMCompositeModelParams**, and uses these to train an internal PatMax model for run-time pattern matching.

You can query the number of component models added to the Multi-Model by calling the **ccPMMultiModel::numModels()** function, and you can get a list of their IDs by calling the **ccPMMultiModel::getModelIds()** function.

Next, you specify the run-time parameters for the Multi-Model including the run-time operating mode and related parameters described in the following section, and then run the Multi-Model on your run-time images.

The Multi-Model run-time parameter set, specified as a **ccPMMultiModelRunParams** object, is an extension of the PatMax parameter set. Thus, the complete search functionality of a PatMax model is available to the Multi-Model user.

Operating Modes

A **ccPMMultiModel** can be run in various modes to improve efficiency by exploiting any prior knowledge of the likely sequence of appearances.

There are four modes of operation defined by **ccPMMultiModelRunParams::mode()**:

- Sequential:

  Models are run in a predefined order (queue) until enough results are found with scores greater than or equal to a user-defined confusion threshold to satisfy the PatMax **ccPMMultiModelRunParams::numToFind()** run-time parameter. You define the confusion threshold with **ccPMMultiModelRunParams::confusionThreshold()**. A separate run-time parameter, **ccPMMultiModelRunParams::useXYOverlapBetweenModels()** determines whether lower scoring overlapping results are discarded and are therefore not counted towards the total applied to the **ccPMMultiModelRunParams::numToFind()** threshold. A further run-time parameter, **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** can be used to allow results from only one PatMax model (within the PatMax Multi-Model) to be counted towards the **ccPMMultiModelRunParams::numToFind()** threshold.

  This is intended to address the use case where the application knows that it wants to find/count multiple parts and the parts may be of various types, for example, type A, type B, type C, and so on. In addition, the application has prior knowledge that at any one time the field of view will contain parts of only one type. The sequence of the type, however, cannot be predicted with certainty.

  The order of model execution (queuing sequence) is settable using **ccPMMultiModel::modelIdQueue()** and this may be done after each run-time call, if so desired. If no models score greater than or equal to the **ccPMMultiModelRunParams::confusionThreshold()**, then the single model results from the highest scoring model are returned.

- Sequential Most Recently Successful:

  A variant on sequential mode, where after each call to run the successful model is promoted to first in the queuing sequence.

- Sequential Most Successful:

  A variant on sequential mode, where after each call to PatMax Multi-Model run-time, a histogram tabulating the successful model id's of the previous N calls run-time is updated (N is an integer you define that provides the temporal window over which model success is evaluated). N is defined by

ccPMMultiModel::resultStatisticWindowLength(). The queuing sequence is then set to reflect the relative values in this histogram. You can reset this histogram using **ccPMMultiModel::resetResultStatistics()**.

- Exhaustive:

  All models in a Multi-Model are executed simultaneously (using as many processor cores as are enabled by the work manager – see *ch_cvl/workmgr.h*). The reported result set includes single model results either from all models (if **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** is false) or just a single model (if **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** is true). The results are sorted in order of highest score first. The component model used to generate each individual result is identifiable through the API.

You have write access to the model queuing sequence and the results returned by the Multi-Model will identify which of the models generated each result. This allows you to implement your own sequence modifying heuristic, which can be based on prior results or any other data. For example, the application may decide to run all the models (in the Multi-Model) on a representative set of run-time images, then base the queuing sequence on the relative scoring of the models.

The application of the PatMax run-time **ccPMMultiModelRunParams::xyOverlap()** parameter to results originating from different PatMax models may be enabled or disabled for all run-time modes through the run-time API.

Example for Run-Time Modes and Results

To illustrate the interaction between the run-time mode and the **ccPMMultiModelRunParams::numToFind()** and **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** run-time parameters, consider the following example.

A **ccPMMultiModel** contains two PatMax models trained for two pattern types: A and B. The **ccPMMultiModel**'s queuing sequence is set so that A will be tried before B when running in sequential mode. The **ccPMMultiModelRunParams::confusionThreshold()** is set to 0.80. In the current field of view there are two instances of type B.

The component PatMax models, if run separately, would return scores as follows:

```
A1=0.92

B1=0.98

B2=0.88
```

The following table shows which results are returned with the various allowed combinations of **ccPMMultiModelRunParams::mode()**, **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** and **ccPMMultiModelRunParams::numToFind()** run-time parameters.

Mode	Sequential						Exhaustive					
reportResultsFromOneModelOnly	false			true			false			true		
numToFind	1	2	3	1	2	3	1	2	3	1	2	3
Results	A1	B1 A1 B2	B1 A1 B2	A1	B1 B2	B1 B2	B1	B1 A1 B2	B1 A1 B2	B1	B1 B2	B1 B2

Note: It may seem surprising that the Multi-Model in some cases returns 3 results when **ccPMMultiModelRunParams::numToFind()** has been set to 2. This occurs because the Multi-Model runs its component PatMax models with this same parameter value (that is, **ccPMMultiModelRunParams::numToFind()** set to 2). If **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** is set to false, then the Multi-Model reports all the results generated by its component models that reach the accept threshold (except the cases where the **ccPMMultiModelRunParams::xyOverlap()** is enabled and pertinent).

As described previously, the **ccPMMultiModelRunParams::confusionThreshold()** is used to terminate the search (that is, when the number of results scoring greater than or equal to the **ccPMMultiModelRunParams::confusionThreshold()** reaches **ccPMMultiModelRunParams::numToFind()**, then no more component models are run). However, the number of results returned is governed by the accept threshold, so depending on the setting of

ccPMMultiModelRunParams::reportResultsFromOneModelOnly(), all results scoring above the accept threshold will be returned for either the best run component model or all the run component models (note, **ccPMMultiModelRunParams::useXYOverlapBetweenModels()** is respected in this latter case).

Using PatMax

This section describes how to train PatMax from a shape description or image, and how to use the trained tool to find likenesses of the pattern in run-time images.

Shape Training

Shape training allows you to train PatMax directly from a model derived form the **ccShape** class. The use of shape descriptions instead of images offers several significant advantages:

- Shape training allows you to define the optimal model to train. For example, if your application needs to locate a fiducial mark but all the fiducial marks encountered by your application have slight pattern variations and defects, then any image you use to train a pattern will include features that are not present in other images. Instead of training a pattern from an acquired image of a fiducial mark, you can create a shape description of an ideal fiducial.
- Shape training is not corrupted by noise.
- Shape training allows you to specify the model origin precisely.
- Shape training allows you to select the portion of the object you need to model more easily than masking a training image.
- Shape training is more effective in those situations where there is a wide range of scale changes present in the run-time images.
- Shape training allows PatMax to explicitly use information about the locations of corners in the model.
- Shape training requires less memory than image training.

When you train PatMax using a shape description you pass it a reference to a **ccShape** object that models the object to be located. Cognex provides an extensive library of primitive shapes for you to use such as rectangle, circle, line, ellipse, point, and others. You can also create your own primitive shapes using the **ccGenPoly** class.

While all of these primitive shapes are available, most patterns you will need to train are more complex and require multiple primitives combined in various ways to represent the desired pattern. To create these complex models Cognex provides shape tree classes which are container classes that can hold many primitives, related in various ways, to represent one model. The shape tree classes also derive from **ccShape** and you will normally pass your shape description as a pointer to a shape tree rather than a pointer to a primitive. The primitive shape classes and the shape tree classes are covered in the *Shapes* chapter of the *CVL User's Guide*, where you will also find information about building shape trees.

Flattening and Connecting Shape Trees Before Training

When you construct a shape using a polygon, generalized polygon, contour tree, and so on, the shape connections are explicit. Shape trees created other ways, for example, from imported DXF files, often contain no explicit information regarding the connectivity of shapes. A rectangle might be represented as four line segments whose endpoints are coincident. The shape tree *flattening* and *connecting* operations interpret these coincident shape endpoints and make such connections explicit. For example, flattening and connecting a rectangle imported as a DXF file could result in a contour tree containing the four line segments.

When you train PatMax with explicit information you will in most cases achieve better alignment performance. PatMax should be able to train from any shape you give it and there will be no throws if you don't flatten and then connect. Flattening and connecting is only recommended if you have a shape that does not explicitly represent the connections

between contours that are actually connected. Without this explicit representation, PatMax will not be able to use the connection information to disambiguate candidate poses.

See also the *CAD File Import* chapter of the *CVL User's Guide*.

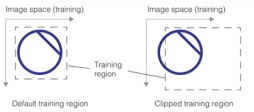
The *clientFromImage* Transform

When you use shape training, the geometric description is given in client coordinates, the coordinate system PatMax uses for all calculations. To inform PatMax about how these client coordinates relate to pixels you also provide a *clientFromImage* transform it uses to relate the training pattern to run-time image patterns.

clientFromImage transform, shape training

The Training Region

When you use shape training you can also specify a training region in training image space. See the example in *clientFromImage transform, shape training* on page 304. If you do not specify a training region PatMax supplies a default which is the bounding box that completely encloses the shape description. If you do specify a training region be aware that any part of the shape description that does not lie inside the training region will be clipped and the clipped part will not be trained. See the figure below.



Training region examples

As long as the shape description lies completely within the training region, the exact rectangle is only important if *scoreUsingClutter* is true when the tool is run, since the rectangle represents the region in the run-time image over which the clutter is computed.

Which Features Are Trained

For shape training you also provide a training region which is a rectangular area in image space. PatMax maps the shape description into image space and applies the training region rectangle. Only features that fall inside the training region are trained. See the *clientFromImage transform, shape training* on page 304 figure in the *clientFromImage* Transform section.

Note that only portions of the shape within the training region that can be resolved reliably within the pixel grid defined by the client transform and the granularity setting will be trained as part of the pattern. For example, sharp corners, small circles, and close parallel lines with opposite polarity may not contribute at all to the pattern features. This is especially true of coarse features since these are typically trained at significantly higher granularity (lower resolution) than fine features. If no reliable features exist, PatMax will issue an error stating that there are insufficient features to train. This may occur if the entire training shape or region is very small (for example, 20x20 pixels or smaller) and/or the shape is highly detailed relative to the training resolution. This problem can often be solved either by using a lower granularity setting (which will require turning off grain auto-selection), or by using a client-from-image transform with a smaller pixel size.

Assigning Weights

When you train PatMax with a shape description, the shape is generally composed of many primitive shapes encapsulated in a shape tree object that describes the complete shape you wish to find in run-time images. Each of these primitives can be assigned a weighting factor in the range -1.0 through +1.0 that is used at run time to compute a score for how well the trained model matches the likeness found. Note that the effective weight of any primitive is actually assigned an implicit weight and the weight of its ancestors (see the *Shape Models* chapter in the *CVL User's Guide*).

All of the primitives (including those with 0 weight) are used for training. At run time, however, primitives are treated differently depending on their effective weight as follows:

- Positive effective weight:
  Results favor those poses for which the primitive matches the image features with high accuracy, although the magnitude of the weight has no effect on accuracy. The primitive contributes positively to a PatQuick result score in an amount proportional to the weight, to the primitive perimeter, and to the degree of match at the result pose. Image features that match the primitive are not considered clutter. Such features instead contribute to the coverage score, but the magnitude of the weight is not used when computing coverage.
- Negative effective weight:
  Results favor those poses for which the primitive does not match the image at all. The primitive has no effect on the accuracy of the returned pose. The primitive contributes negatively to a PatQuick result score in an amount proportional to the weight, to the primitive perimeter, and to the degree of match at the result pose. Image features that match the primitive do not contribute to the coverage score. Such features are instead considered clutter.
- Zero effective weight:
  The primitive has no effect on the result pose or its accuracy, and contributes nothing to the PatQuick result score, regardless of degree-of-match. Image features that match the primitive do not contribute to the coverage score. Such features are instead considered clutter.

Because the magnitude of a weight does not affect the coverage or clutter scores, a PatMax algorithm result score is only affected by the sign of weights (which determines whether matched features contribute to clutter or coverage, as described above). Therefore, if you wish to experiment with weighting schemes in order to optimize performance with respect to the accept threshold, the PatMax algorithm (if used) should be temporarily disabled. The result score returned will then be the PatQuick result score, which is affected by the magnitudes of the weights. When the PatMax algorithm is again enabled, the PatQuick score will continue to be computed internally and compared to the accept threshold, but it will in general no longer be returned as the score of the results (the PatMax algorithm score will be returned instead).

Training With Wireframes

Prior to CVL 6.0.2 PatMax shape training could be performed using only **cc2Wireframe** objects. This API has been deprecated and is now maintained for backward compatibility only. The new training API accepts **ccShape** objects which include wireframes as well as many other shapes.

cc2Wireframe is a special case of a **ccGenPoly** that includes segment length tolerance information. Since PatMax does not use tolerance information, most new PatMax users who wish to use wireframe models will use **ccGenPoly** or **ccGenPolyModel** instead of **cc2Wireframe**. New users who need segment length tolerance information for tools other than PatMax will likely use **cc2Wireframe** or **cc2WireframeModel**.

If you have older applications that you wish to change to use the new PatMax training API, and you wish to keep your wireframe models, you need to put your wireframe shapes into a shape tree prior to training. This requires making some minor code changes. For example, your wireframe code probably looks something like this if you use weights:

```
pattern.train(wireframeVector, weightVector, clientFromImage,
 algorithms);
```

If you do not use weights it probably looks like this:

```
pattern.train(wireframeVector, clientFromImage, algorithms);
```

```
The equivalent new code will look like this:
```

```
ccGeneralShapeTreeModel model;
for (int i = 0; i < wireframeVector.size(); i++)
{
 const cc2Wireframe& wf = wireframeVector[i];
 double w = 1.0;
 // The next line is necessary only if weights are used
 w = weights[i];
```

```
 model.addChild(new ccGenPolyModel(wf,
 ccShapeModelProps(wf.isPolarityReversed(), w)), true);
}
pattern.train(model, clientFromImage, ccPelRect(0,0),
 algorithms);
```

Note above, that setting weights is only required if you use weights.

Image Training

For image training PatMax uses all of the information in the pattern training image you supply. You should avoid including features in the training image that will not be present in the run-time image.

In order to train a pattern from a pattern training image, the image must contain distinguishable features. For best results, you should observe the following guidelines when selecting a training image:

The pattern should include both coarse and fine features.

The pattern should include information that will let PatMax distinguish instances that vary in all enabled degrees of freedom. The table below shows examples of patterns that might lead to degenerate results (as described in the section *Degenerate Results* on page 293.

Degree of Freedom	Potentially confusing patterns	Less confusing patterns
Scale		
Rotation		

Patterns with potentially degenerate results for scale and rotation

PatMax can return diagnostic information at training time that indicates whether a trained pattern is potentially degenerate in a particular degree of freedom. See *Pattern Training Information Strings* on page 310.

Training Image Coordinate System and Calibration

If the training and run-time images have different coordinate systems or calibrations, then identical patterns in the run-time images will be found at different scale or angle than the trained pattern.

For example, if the same optical and mechanical configurations are used to acquire training-time and run-time images, but the training-time image client coordinate system units are one half the size of the run-time image client coordinate system units, the pattern in the run-time image will be found at a scale of 0.50.

You should make sure that the same calibration has been computed for both pattern training and run-time images. If you have not calibrated the pattern training and run-time images, then you should make sure that the optical and mechanical configurations used to acquire the images are the same.

Which Features Are Trained

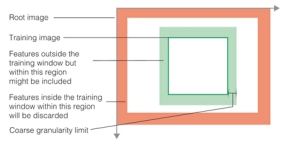
PatMax uses pixels from the root image upon which the pattern training image is based for pattern training. This means that PatMax will train all features found inside the pattern training image window, but it can cause PatMax to detect features outside the pattern training image. The two effects of this change are

- If the pattern training window is close to the edge of the root image, feature boundary points that are closer to the edge of the *root image* than the coarse granularity limit may not be detected.

  If the pattern training window is not close to the edge of the root image then it is guaranteed that every feature inside the window is detected, regardless of the coarse granularity setting.

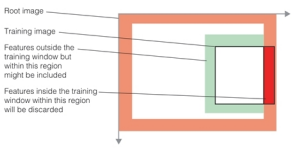
- PatMax can detect features that are outside of the pattern training image if those features are within the coarse granularity limit of the edge of the pattern training window *and* if they are not within the coarse granularity limit of the edge of the root image.

Pattern rotation with origin at pattern center on page 310 summarizes these two effects.



Features outside the training image can be considered

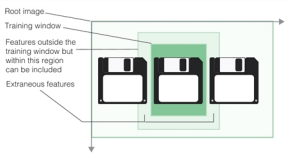
If the edge of the training image is close to the edge of the root image, then features within the training image might not be trained, as shown in the figure below.



Features within the training image can be discarded

Because PatMax may train features that are outside the training image, you should take care that no features that are not part of the actual pattern are present in the root image near the training image window.

The figure below shows a case where unwanted features could be trained even though they are outside of the training window.



Extraneous features trained from outside of training image

In general, you should observe the following basic guidelines:

1. Do not train a pattern that has features that are close to and just outside the training window; these features may get trained in as a part of the pattern.
2. Do not train a pattern with features that are close to and just inside the training window if the window is close to the edge of the root image; these features may not get included in the trained pattern.
3. Always use the diagnostic function to display the feature boundary points trained as part of a pattern; verify that all the features you expected to be trained are trained and that no extraneous features from outside the training image window are trained. See *Diagnostic Displays* on page 315.

Edge Thresholds

By default, PatMax automatically computes and uses edge thresholds when it extracts features from a training image. You can override PatMax's thresholds by specifying a single edge threshold value in the range 0 through 255, where the edge strength is the difference in grey level values across the feature edge.

If you specify an edge threshold for training, edges with strength below the threshold are assigned a random orientation and are not incorporated in the trained pattern.

As described in the section *Run-Time Edge Threshold* on page 312, you can specify a separate edge threshold for run-time images.

All PatMax Training

The following sections apply to both shape training and image training.

Pattern Granularity

For most applications, PatMax does a good job selecting the coarse and fine pattern granularity limits. If you want to override PatMax's granularity limits, you should do so by evaluating PatMax's pattern granularity settings using the training pattern and the diagnostic display. See *Diagnostic Displays* on page 315.

Keep in mind that PatMax's strategy in choosing granularity limits is to choose the largest coarse granularity that detects features that can be reliably used to quickly locate the pattern and to choose the smallest fine granularity setting which will reliably and precisely locate the pattern.

The following are examples of when you might decide to override PatMax's settings:

- If the coarse granularity diagnostic display is including features that appear not to represent actual feature boundaries, you can try increasing the coarse pattern granularity to exclude the redundant small features. This might increase the alignment speed.
- If the fine granularity diagnostic display is including features that are not actually part of the pattern, such as surface texture, you could try increasing the fine pattern granularity to exclude the extraneous small features. This might improve reliability.
- If the fine granularity diagnostic display does not include fine features that are part of the pattern, such as fine teeth on a gearwheel, you could try decreasing the fine pattern granularity to include the small pattern features. This might increase accuracy.

You should not attempt to override PatMax's granularity settings without carefully examining the diagnostic feature display. See *Diagnostic Displays* on page 315. In general, you should choose the granularity that produce features in the run-time diagnostic window that match features the trained model.

Note: When using the PatFlex (or PatPersp) algorithm, the default granularities are generally smaller due to the need for more detail. If you set your own granularities they will also be generally smaller than those you would set for other algorithms.

Advanced Training Mode

In some cases, PatMax's automatic grain limit computation may produce a coarse granularity limit that does not work well with patterns that contain repeating features. In these cases, you can specify *advanced training mode*, which performs additional refinement steps during grain limit computation.

High Sensitivity Mode

PatMax can be run in standard mode or high sensitivity mode. Check your images for contrast and noise. Good quality images should be run in standard mode but if your images have poor contrast or are noisy, you may get better results using high sensitivity mode. Keep in mind however, high sensitivity mode generally requires more execution time than standard mode.

When you use high sensitivity mode you can also set the *sensitivity parameter* which allows you to specify your image quality. The sensitivity parameter is a number in the range 1.0 through 10.0 where lower numbers specify better quality images and higher numbers specify poor quality images. The default is 2.0. You may need to experiment with this parameter range to see which setting produces the best results for your images. See the discussion in the section *High Sensitivity Mode* on page 282.

Polarity

By default, PatMax will only find matches where the trained pattern polarity and the run-time image pattern polarity are the same. You can specify that PatMax ignore the polarity of patterns in the run-time image. If you specify that PatMax ignore pattern polarity in the run-time image, you will increase the number of potential matches, which can increase image confusion. Also, ignoring pattern polarity reduces the search speed.

Note: You can change the value of this parameter at training time or run time with almost no execution time penalty, although the effect of changing the value can affect search speed.

The default for trained patterns specifies the darker side of a pattern boundary is negative and the lighter side of the boundary is positive. For example, see the figure below.

Enlarged portion of a trained pattern showing default polarity

For shape descriptions you can reverse the polarity by setting **ccShapeModelProps::isReversedPolarity()** to true. Polarity for shape descriptions is covered in more detail in the **ccShapeModel** and **ccShapeModelProps** reference pages.

Pattern Origin

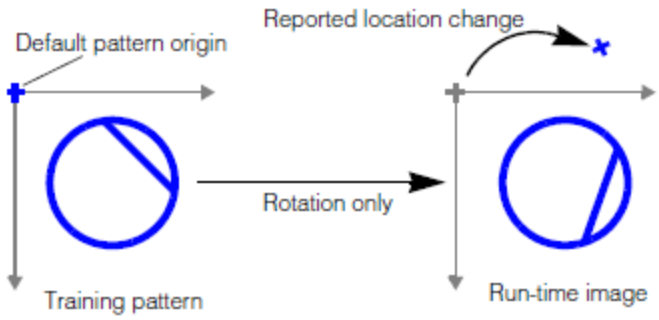
As described in the section *Generalized Pattern Origin* on page 284, PatMax lets you specify either a simple pattern origin point or a generalized pattern origin. This section provides guidelines for both origin forms.

Simple Pattern Origin

By default, PatMax returns the position of a found instance of the model in a run-time image as the position that corresponds to the origin of the training pattern client coordinate system. You can specify a different location within the model for PatMax to use when reporting locations.

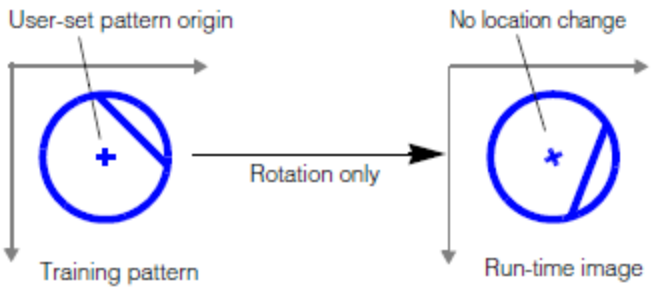
Note: If your programming interface does not support client coordinate systems, the default pattern origin is the origin of the training pattern.

If you use the default pattern origin location, keep in mind that if you enable any generalized degree of freedom, instances of the pattern might be reported at a shifted location. The figure below shows an example where the transformation between the two patterns is rotation only, but because the default origin is used, a translation is reported in addition to the rotation.



Rotation causes apparent translation of default origin

The figure below shows the effect of specifying a pattern origin at the pattern center.



Pattern rotation with origin at pattern center

In general, you should either

- Set the origin to correspond to a point of interest within the trained pattern.
- Set the origin to be at the center of the trained pattern.

Generalized Pattern Origin

You specify a generalized pattern origin by supplying a transformation object at training time. PatMax applies this transformation to the training pattern before attempting to locate the pattern in a run-time image, and it returns results relative to the transformed training pattern.

In general, you supply a generalized pattern origin to compensate for known scale or rotation of the training pattern. For more information on using a generalized pattern origin, see the section *Generalized Pattern Origin* on page 284.

Pattern Training Information Strings

When you train a pattern using PatMax, PatMax may provide diagnostic information about the pattern in the form of text strings. To check for this information call **ccPMAlignPattterm::infoStrings()** following training to read any text strings the tool writes. The table below describes the messages that PatMax can produce at training time. For information about the diagnostic displays mentioned in this table see *Diagnostic Displays* on page 315.

Information Strings Messages	Notes
10000: Pattern may contain too few features.	PatMax did not detect enough features to train a reliable pattern. Examine the diagnostic display.
10001: Difficult to choose feature granularity; manual override may be desirable.	Examine the diagnostic display. This message is only returned if you have configured PatMax to determine feature granularity automatically.
10100: Pattern may be less accurate than expected because the training pattern appears blurry.	PatMax trained a pattern, but because the pattern was blurry, the accuracy of alignment results may be reduced. This message is only returned if you are training a pattern using the PatMax algorithm.
10101: Pattern may contain insufficient information to measure *DOF* reliably.	The trained pattern is potentially confusing in the indicated degree of freedom. See the section *Image Training* on page 306 for information on confusing patterns.
10102: Pattern is degenerate. The results will probably be unstable because all of the coarse granularity boundary points have the same direction.	At the coarse granularity limit, all of the feature boundary points have the same orientation. Examine the diagnostic display.
10200: Pattern may run slowly due to predominance of fine features; manual override may be possible.	Because of a large number of small features in the pattern, alignment may take more time. Try manually specifying a larger coarse granularity value.

PatMax training-time information strings

PatMax Customization Packs

PatMax provides support for loading Cognex-supplied PatMax Customization Packs (PCPs). These PCPs configure PatMax for specific application types. The effect of loading a PCP is to change certain internal PatMax parameters.

Note: You must load a PCP *before* training a pattern.

PatMax Alignment Guidelines

This section provides an overview of the parameters you specify when you use PatMax to search for a model in a run-time image.

In general, you should keep the following basic guideline in mind: The more specific the information that you can provide about the objects in the run-time image, the faster PatMax will operate. Each of the following items increases the size of the run-time space and increases the amount of time PatMax requires to find objects:

- Increased number of enabled degrees of freedom
- Larger zones for enabled degrees of freedom
- Decreased score threshold

Constraining the overall size of the run-time space can also tend to reduce the likelihood that PatMax will encounter degenerate results.

Degrees of Freedom

For each of the possible degrees of freedom other than location you must either specify a nominal value, or a range of acceptable values. PatMax only searches within the combination of nominal and ranges that you specify.

In general, you should enable a degree of freedom and specify a zone if you expect patterns to vary in the degree of freedom or if you need to measure the degree of freedom.

Score Threshold

The higher a score threshold you specify, the faster PatMax will be able to perform an alignment. You should establish the score threshold for your application by running a series of test alignments. Select a score threshold that is slightly below the lowest score that an actual instance of the pattern receives.

A good starting point for the score threshold is 0.5.

Number to Find

You should specify the number of results that you expect the run-time image to contain. Because of PatMax's ability to find transformed versions of the trained pattern, run-time images may contain many more instances of a pattern than you might expect. Keep in mind however, that as you increase the number of results, you also increase the run time.

If there is only one instance of the pattern in the image, request only one result. For most alignment applications, you should specify one result.

Elasticity

Unless you expect pattern variations that cannot be described using a linear transformation, you should specify a value of 0.0 for elasticity.

If you experience alignment failures or unstable score results, you can begin experimenting with low elasticity values. In general, you should specify the lowest elasticity value that provides stable consistent alignment results.

If you are expecting significant nonlinear pattern deformation, you should use the PatFlex algorithm. If the only deformation type you are expecting is planar perspective distortion, for best speed performance, you can use the PatPersp algorithm. See *Non-Linear Pattern Deformation* on page 288.

Scoring Clutter

If objects will appear on a variety of backgrounds, you should specify that PatMax ignore clutter when scoring objects.

If your application is an alignment application in which the background does not change, you should specify that PatMax consider clutter when scoring objects.

Extreme Scale Changes

PatMax can find patterns at a wide range of scale changes between the trained model and the run-time images. Increasing scale changes in the run-time image can affect PatMax performance and accuracy. You should keep in mind the following points about the effect of scale changes on PatMax:

- The presence or absence of scale change itself has no effect on PatMax speed or accuracy.
- The effect of increasing scale change is extremely image-dependent.
- If you are searching for extremely small, simple patterns, train a larger pattern, preferably using a shape description.

As the size of the scale change between the trained pattern and the run-time pattern instance becomes very large, several factors come into play. These factors are different depending on whether your application uses patterns trained from acquired images or patterns trained from shape descriptions.

Note: All of the numbers discussed in this section are approximate. The specific performance of PatMax and the effect of specific scale changes is dependent on the content of the images used for pattern training and pattern location. You should conduct your own tests using a range of images to determine the best PatMax parameter settings for your application.

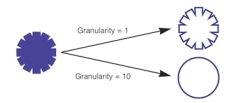
Patterns Trained From Acquired Images

As the size of scale change between the trained pattern and the run-time pattern increases, PatMax's performance can begin to degrade. In general, scale changes below 10% to 20% have no effect on PatMax's ability to discriminate patterns within a run-time image, and no effect on PatMax's ability to locate patterns with ultrahigh precision (better than 0.10 pixels).

As scale change increases above 10% to 20%, *some* applications will experience a reduction in accuracy levels.

At extremely large scale changes (greater than a factor of 2), *some* applications will experience problems discriminating patterns in run-time image.

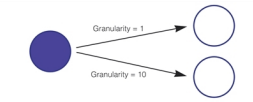
PatMax is more likely to be able to successfully handle large scale changes when the overall shape of the trained pattern does not change at different granularities. The figure below shows an example of a pattern whose shape changes at different granularities.



Pattern shape change with granularity change

The pattern shown in the figure above may be more difficult for PatMax to find at extreme scale change.

The pattern shown in the figure below has the same shape at different granularities. This pattern may tend to work better with extreme scale changes.



No pattern shape change with granularity change

Note: If the run-time pattern is less than about one third the size of the trained pattern, the coverage score it receives will be reduced. This reduced coverage score, in turn, lowers the overall score received by the pattern, even though the shape of the pattern may be a good match with the trained pattern.

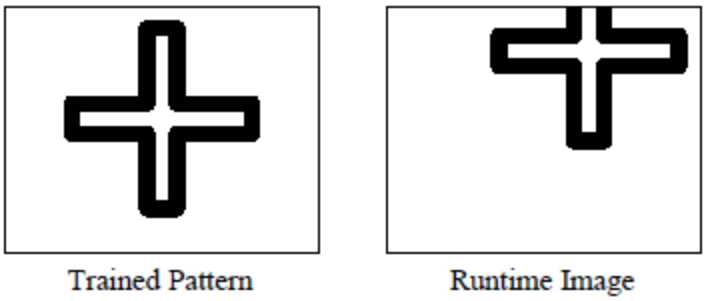
Run-Time Edge Threshold

By default, PatMax automatically computes and uses an edge threshold when it extracts features from a run-time image. You can override PatMax's thresholds by specifying a single edge threshold value in the range 0 through 255, where the edge strength is the difference in grey level values across the feature edge.

Features in the run-time image with edge strength below the threshold that you specify are assigned a random orientation.

Outside Region PatMax

Outside Region PatMax is a new CVL feature added in CVL 6.3. This feature allows you to set up PatMax so that it does not penalize runtime images that extend beyond the search region. For example, the runtime image in the figure below could still receive a coverage score of 1.0 despite part of the image falling outside the search region.



Outside Region PatMax example

Outside Region PatMax is implemented with the function **outsideRegionThreshold()** in **cc_PMRunParams**. Use this function to specify an *outsideRegionThreshold* value, which represents the proportion of the trained pattern that can appear outside the search region without affecting the PatMax result coverage score. Specify the *outsideRegionThreshold* value as a value greater than 0.0 and less than or equal to 0.999. For example, a value of 0.2 allows 20% of the features of the trained model to fall outside the search region without affecting the coverage score.

The following results in **cc_PMResult** are used with this feature:

- **outsideRegionFeatureProportion()** contains the proportion of the pattern features that fall outside the search image.
- **outsideRegionAreaProportion()** contains the proportion of the pattern rectangle that falls outside the search image. This result takes into account the possibly rotated and scaled bounding box of the training region, and does not take pattern masking or runtime masking into account.

Consider the following points when using the Outside Region PatMax feature:

- Outside region mode can be used in conjunction with the PatMax or PatQuick algorithms, but not the PatFlex or PatPersp algorithm.
- Outside region mode works with both image-trained patterns and synthetically-trained patterns.
- PatMax runs two to six times slower when you use outside region mode, depending on the sizes of the pattern and image, and the *outsideRegionThreshold* value.
- If you provide a window for the runtime search area, outside region mode considers the window to be the only valid pels.
- If you provide a runtime mask in addition to specifying an *outsideRegionThreshold* value, the pels specified under the mask as do-not-care are not normalized away when computing the normalized coverage score.
- During the PatQuick phase, the outside region scores may be noisy. In this case, increase the **xyOverlap()** value so that correct results do not get inadvertently pruned.
- Testing at Cognex has found that, for chip wafer scenes, outside region mode works well in conjunction with the high sensitivity mode introduced in CVL 6.2.

Point PatMax

In some cases, your application may already have an accurate coarse location for a pattern instance in a run-time image. For applications such as this, PatMax allows you to specify a *start pose* defining the coarse location of the pattern instance (including its size, rotation, aspect ratio, and translation. When you use Point PatMax, the tool does not perform the PatQuick coarse alignment phase. Instead, it refines the start pose that you supply to sub-pixel accuracy.

Point PatMax only refines the start pose for degrees of freedom where both of the following are true:

- That degree of freedom is enabled at run time.
- That degree of freedom has a nonzero range.

If you use Point PatMax, keep in mind that the actual pattern instance in the run-time image must be very close to the start pose that you specify.

PatFlex Guidelines

You should only use the PatFlex algorithm under the following conditions:

- You expect significant nonlinear deformation of the trained pattern in your run-time images.
- You wish to produce an undeformed version of the run-time image.

The guidelines in this section can help you use the PatFlex algorithm effectively.

Use Appropriate Image Content

PatFlex works best when used to locate patterns with the following characteristics:

- A range of feature sizes
- A variety of feature shapes and orientations

You should avoid using PatFlex with patterns that are dominated by regular arrangements of straight lines.

Limit the Number of Control Points

Increasing the number of control points tends to increase the amount of time required to refine a PatFlex result. In most cases, PatFlex will return accurate results if you specify six control points in both the x- and y-direction.

Limit Degrees of Freedom and Deformation Rate

As is the case with the standard PatMax and PatQuick algorithms, increasing the number of enabled degrees of freedom or the zone size for any enabled degree of freedom increases the amount of time required for the search. The expected deformation rate (specified at run time) has a similar effect on performance; the larger the expected deformation rate, the longer the search may take. Specifying a larger deformation rate for the trained pattern can increase training time but has only a small effect on search time.

Pattern Origin

PatMax will return a more accurate pose if you locate the PatFlex pattern origin inside the pattern. When you create your pattern model, choose the model origin somewhere within the pattern boundaries.

PatPersp Guidelines

You should use PatPersp if the only deformation type you are expecting is planar perspective distortion. Guidelines for PatPersp are the same as for PatFlex except that you cannot set the PatFlex-only run-time parameters (specified in **ccPMFlexRunParams**) for PatPersp.

Using Nonlinear Client Coordinates

If you supply a run-time image that has a nonlinear client coordinate system, PatMax computes the best-fit linearization of the client coordinate system across the entire input image, then applies that transformation to the features in the input image before locating the trained pattern.

After PatMax locates the pattern instance or instances in the input image, it computes the best-fit linearization of the nonlinear client coordinate system at the origin of each found pattern instance, and it maps the returned pose through that local linearization before returning it.

For more information, see the chapter *Using CVL Vision Tools* in the *CVL User's Guide.*

Run-time Information Strings

When you search for a pattern using PatMax, PatMax may provide diagnostic information about the pattern in the form of text strings. To check for this information call one of the following functions after the search to read any text strings the tool writes.

ccPMAlignPattterm::infoStrings()
ccPMAlignResult::infoStrings()
ccPMAlignResultSet::infoStrings()

The table below describes the messages that PatMax can produce at run time.

Information Strings Messages	Notes
20300: *N* results were discarded due to contrast threshold. The score of the best discarded result was *S*.	PatMax discarded one or more patterns because their contrast was below the contrast threshold you specified for the alignment. You may be able to see these results by lowering the contrast threshold. Note that the score value included in this message is approximate, since the result was excluded from processing before the final score was computed.
20301: *N* results were discarded due to proximity to stronger results. The score of the best discarded result was *S*.	PatMax discarded one or more patterns because they overlapped other better matches. You may be able to see these results by adjusting the relevant overlap threshold. Note that the score value included in this message is approximate, since the result was excluded from processing before the final score was computed.
20302: *N* results were discarded due to excess clutter. The score of the best discarded result was *S*.	PatMax discarded one or more patterns because they had too many extraneous features. You may be able to see these results by ignoring clutter when scoring. This message is only returned if you have configured PatMax to consider clutter when scoring pattern instances.
20303: Degenerate system detected for this result.	This PatMax result was one of several degenerate results. You can only obtain this message by requesting the diagnostic message for an individual result, not for an entire set of results.

PatMax run-time information strings

Diagnostic Displays

When you train and run PatMax you have the option to provide a diagnostic object (**ccDiagObject**) and a set of flags. The flags you set cause PatMax to save diagnostic information in the diagnostic object during execution. Later you can run a diagnostic utility to observe the diagnostics for troubleshooting and evaluating your application. PatMax can save detailed information about training, the run-time image, and the results. Be aware that computing and retaining this information increases memory usage and makes your application run slower.

The diagnostic flags are described in the **ccPMAlignPattern** reference page and defined in **ccDiagDefs**. See *dFlags* in **ccPMAlignPattern::train()** and **ccPMAlignPattern::run()**. To cause diagnostic information to be saved add code similar to the following to your application initialization:

```
ccDiagObject diagObj;
c_UInt32 diagFlags=ccDiagDefs::eRecordDefault;
```

This code creates a diagnostic object and sets diagnostic flags that enable recording tool inputs and final results.

To observe recorded diagnostics add the following code to your application somewhere after running PatMax:

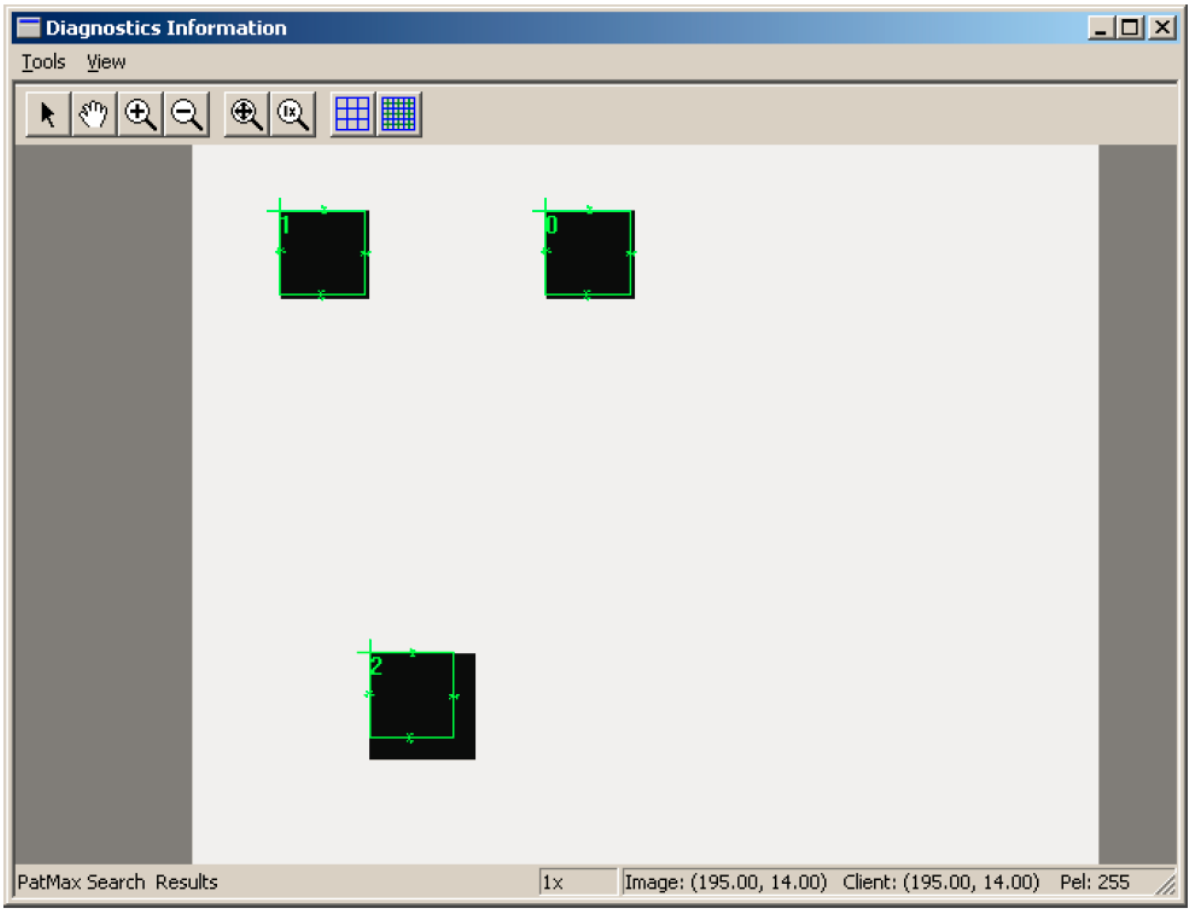
```
ccDiagServer::init() //Initialize the server thread
ccDiagServer::showDaigObject() //Display the diagnostics
ccDiagServer::exit() //End the thread
```

To demonstrate the use of diagnostic displays we have added the code described above to the *pmalign2.cpp* sample code included with your CVL release. The table below summarizes how the sample code operates before it is modified to display diagnostics.

Program steps	Display windows	Control windows
1. Create a run-time image containing three target patterns		
2. Create a training image containing a square wireframe pattern		
3. Train the tool	**Trained Patquick Pattern** Displays the pattern used for training **Trained Patquick Pattern** Displays the trained pattern	**Model** **Click OK to continue** **Trained Features** **Click OK to continue**
4. Run the tool	**Cognex Standard Output** Displays the tool run time, number of patterns found, and the results **Found PatQuick Pattern** Displays the three found patterns, each overlaid with the training pattern (See the figure below)	**Results** Click OK to continue
5. End the program		**cvlproj** Sample Code Complete

pmalign2.cpp sample code operation

The figure below shows the graphic results which display the input image with the trained pattern overlaid onto each found pattern instance. The results are labeled 0, 1, and 2.



PatMax results

To add diagnostic displays to this sample program we need only change one line of code from -

```
c_UInt32 diagFlags=0;
```

to -

```
c_UInt32 diagFlags=ccDiagDefs::eRecordDefault;
```

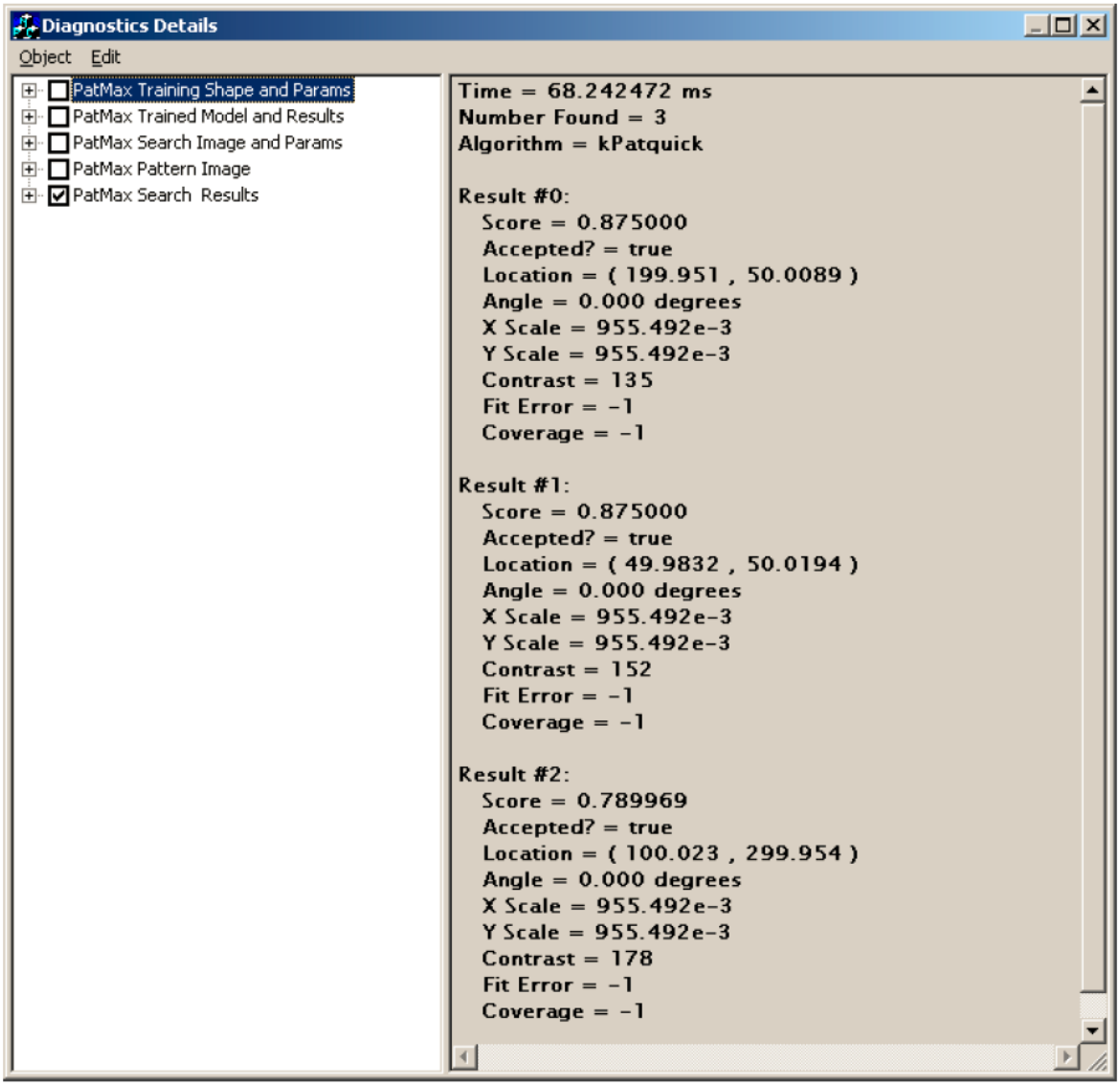
Other required code mentioned above is already part of *pmalign.cpp*.

When we recompile and rerun the program it now functions as show in the table below. Note that the operation is the same as before until Step 4 when we run the tool.

Program steps	Display windows	Control windows
1. Create a run-time image containing three target patterns		
2. Create a training image containing a square wireframe pattern		
3. Train the tool	**Trained Patquick Pattern** Displays the pattern used for training **Trained Patquick Pattern** Displays the trained pattern	**Model** **Click OK to continue** **Trained Features** **Click OK to continue**
4. Run the tool	**Cognex Standard Output** Displays the tool run time, and the number of patterns found **Diagnostics Information** Displays the three found patterns, each overlaid with the training pattern **Diagnostics Details** Displays diagnostic information in two side-by-side panels.	**Diagnostics** Click OK to continue
5. End the program		**cvlproj** Sample Code Complete

pmalign2.cpp sample code operation

Two new windows now appear in Step 4; **Diagnostics Information** and **Diagnostics Details**. The left-hand panel in the **Diagnostics Details** window contains the directory of the stored diagnostic information. When you select an item from the hierarchy, stored text information about that item is displayed in the right-hand panel and stored graphics diagnostics are displayed in the **Diagnostics Information** window. The default display is shown in the figure below which displays the run-time results (**PatMax Search Results**).

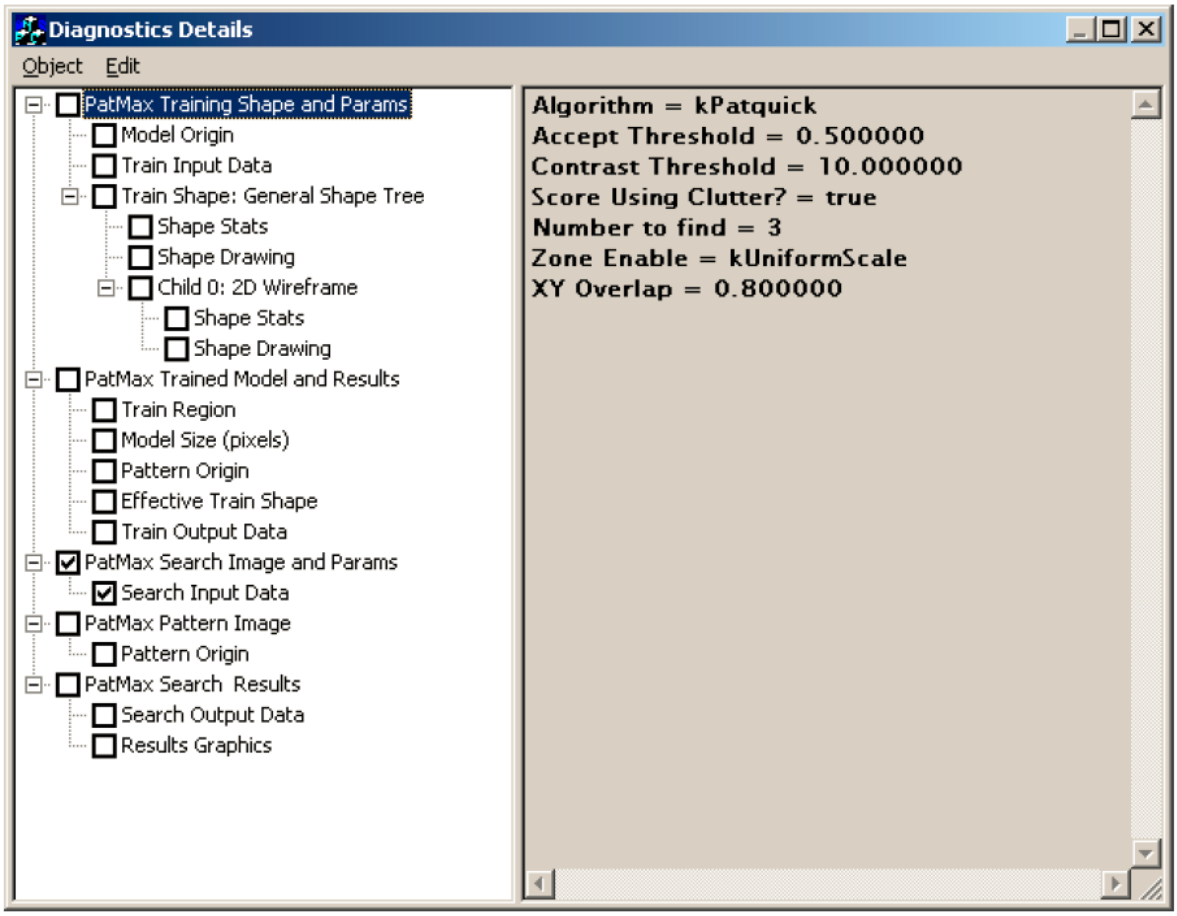


Diagnostic run-time results display

You click on the + and - boxes to open and close the hierarchy, and you click on the adjacent detail boxes to display the detail information. For this example, we used the default diagnostic flags which include the following:

```
eRecordDefault = eInputs | eResults | eRecordOn;
```

This causes the tool to save diagnostic information for the inputs and the results. In the **Diagnostic Details** display above the inputs are included under the first four boxes and the results are included under the last box. Intermediate results can also be saved by setting the `eIntermediate` flag which we did not include in this example. The figure below shows the complete diagnostics hierarchy.



Complete diagnostics hierarchy

Troubleshooting Tips

If you experience problems using high sensitivity mode, it may help to check the training details. (See *High Sensitivity Mode* on page 282 for a description of high sensitivity mode). Training details are part of the training and run-time diagnostics you can record when running PatMax. A general discussion of using these CVL diagnostics is included in the section *Diagnostic Displays* on page 315.

PatMax training (both synthetic and image training) provides training details as part of the CVL diagnostics if you set the diagnostic flags to include both *ccDiagDefs::eIntermediate* and *ccDiagDefs::eRecordOn*. This training diagnostic shows which features will be used to register the pattern for both coarse and fine granularities. If the pattern has curved or short sides, only a small fraction of the features may be used. This behavior can be exacerbated in high sensitivity mode, and even more so if the sensitivity parameter is greater than 2.0.

Visually inspect the training details diagnostic to verify that the trained features appear to be sufficient. If the training details appear deficient, try to improve the selection of training features by manually specifying the coarse/fine granularities and/or reducing the sensitivity parameter value.

Optimizing PatMax Performance

By configuring the PatMax alignment parameters appropriately, you can achieve the optimum balance between execution speed and robustness for your application. The table below lists the factors that affect PatMax execution speed.

Factor	Effect	Notes
Additional degrees of freedom	Each additional degree of freedom you enable increases the required alignment time.	Only enable a degree of freedom if • You expect patterns in your run-time images to vary in this degree of freedom by more than a small amount or • You need to measure this degree of freedom.
Increasing zone size for a degree of freedom	The larger the range of values you specify for a degree of freedom, the longer a PatMax alignment requires. The increase in execution time might not be linearly related to the increase in zone size.	PatMax might return pattern instances that are slightly outside of the range you specify. PatMax only does this if it can do so without increasing the alignment time or decreasing the result accuracy.
Decreasing score threshold	The lower the score threshold you specify, the longer a PatMax alignment requires.	You should determine the score threshold using test alignments.
Number of instances to look for	The more instances you tell PatMax to look for, the longer the alignment may take.	Increasing the number of instances can increase the number of potential matches that PatMax must evaluate.
Run-time area	The larger the run-time area, the longer the alignment takes.	Depending on your application, reduce the size of the run-time area as much as possible.
Pattern size	In general, the larger the size of the trained pattern (in pixels), the *faster* alignments will run.	Select a training pattern that includes as many pattern features as possible. The maximum size of PatMax patterns is 32769 pixels in width and height.
Image confusion	The more confusing the image, the longer alignments will take.	Train patterns that are unique within run-time images.
Coarse granularity	In general, the larger the coarse granularity limit, the faster alignments will run.	Use caution in manually changing the coarse granularity limit.
High sensitivity mode	Produces better results for low contrast and noisy images.	Tune the degree of sensitivity with the sensitivity parameter.
Sensitivity parameter	Specifies the amount of pattern noise rejection for high sensitivity mode.	Use the default value for most applications.

Factors affecting PatMax execution speed

Preventing Degenerate Results

You can reduce the likelihood that PatMax will detect degenerate results by following these guidelines:

- Make sure your pattern offers distinct information in all the degrees of freedom you enable in your alignment, as described in the section *Image Training* on page 306.
- Only enable the degrees of freedom that will actually change from run-time image to run-time image. If the value for a degree of freedom will be the same for all images, specify a nominal value for that degree of freedom.
- Make sure that your run-time image does not contain extraneous features.
- Obtain the information strings for each pattern you train and for each alignment you perform. See *Pattern Training Information Strings* on page 310 and *Run-time Information Strings* on page 315.

Common Image and Pattern Variations

The table below summarizes common image variations, the possible effect of these variations on PatMax, and recommended approaches for handling these variations.

Variation	Possible Effect	Possible Remedy
Typical video noise	None	
Severe video noise	• Reduced accuracy • Failure to locate pattern	• Increase granularity limits, especially the fine granularity limit. • Reduce accept threshold. • Use high sensitivity mode. • Use Composite PatMax for training your PatMax model.
Feature polarity reversal	• False matches • Slightly slower speed	• Ignore polarity.
Extremely low image contrast	• Reduced accuracy • Failure to locate pattern	• Increase granularity limits, especially fine granularity limit. • Reduce accept threshold. • Use high sensitivity mode.
Extremely high image contrast (camera saturation)	• Reduced accuracy • Failure to locate pattern	• Reduce accept threshold. • Adjust camera aperture or lighting to produce less saturated images.
Changes in shading	None	
Rotation or size changes	• Reduced speed • False matches	• Enable appropriate degrees of freedom and specify appropriate nominal values or ranges.
Aspect ratio changes	• Reduced speed • False matches • Reduced accuracy	• Enable appropriate degrees of freedom and specify appropriate nominal values or ranges.
Shape changes (change in geometrical arrangement of pattern features)	• False matches • Reduced accuracy	• Increase elasticity value. • Reduce accept threshold.
Missing features	• Failure to locate pattern	• Reduce accept threshold.
Extra features	• Failure to locate pattern • False matches	• Reduce accept threshold. • Score using coverage score only.

Common image variations

PatMax Parameter and Result Summary

The table below summarizes the parameters that you supply to PatMax and the results that PatMax returns to you. Different parameters and results are used and returned for training and alignment.

Type	Item	Notes
Training parameters	Model origin	Location in pattern model to use when returning result locations in run-time images Can be set at run time with minimal time penalty.
	Training model	The pattern of interest
	Elasticity	Amount of tolerance (in pixels) for nonlinear geometric change in feature location Can be set at run time with minimal time penalty.
	Polarity tolerance	Either consider or ignore the polarity information in patterns in the run-time image Can be set at run time with minimal time penalty.
	Feature granularity	Controls the size of features that make up a trained pattern. Must set before training. Do not set manually for most applications.
	High sensitivity mode	Use standard mode for normal images, high sensitivity mode for noisy or low contrast images.
	Sensitivity parameter	Use the default value for most applications.
	Edge threshold mode	Whether to use automatic edge detection threshold computation or the threshold you supply in training. Do not set threshold manually for most applications.
	Edge threshold parameter	The edge threshold you supply.
	Expected deformation rate (PatFlex only)	Expected pattern deformation rate.
	Mask image	Excludes parts of training image from pattern.
Alignment (run-time) parameters	Enabled degrees of freedom	Only enabled transformations are measured
	Nominal transformation value(s) and zone(s)	For each non-enabled degree of freedom, specify a nominal value. For each enabled degree of freedom, specify a range of values (zone).
	Number of results	The number of instances of the pattern you expect PatMax to find in the run-time image
	Contrast threshold	The minimum contrast value for pattern instances
	Edge threshold mode	Whether to use automatic edge detection threshold computation or the threshold you supply during run-time. Do not set threshold manually for most applications.
	Edge threshold parameter	The edge threshold you supply.
	Score using clutter	Whether or not to consider the effect of extraneous features on pattern instance score
	Score (or accept) threshold	Threshold for considering an instance of the pattern as valid
	Mask image	Exclude parts of run-time image from consideration
	Timeout	Maximum amount of run time PatMax use
	PatFlex run-time parameters	Set for PatFlex only. These are specified in **ccPMFlexRunParams**
	Outside region threshold	Enables patterns that extend outside the search region to be found by PatMax or PatQuick without penalizing the score

Type	Item	Notes
Alignment (run-time) results	Location	Location of each instance
	Pose	Description of how instance is mapped from trained pattern to run-time pattern instance
	Angle	The angle of the found result instance relative to the trained pattern
	X and y scale	X-axis and Y-axis scale of the found result instance relative to the trained pattern
	Scores	Indication of degree of fit
	Outside region feature and area proportion	Specifies what proportion of the result pattern's features or area fall outside the runtime search region
	Accepted	Whether the score received by this result instance is greater than or equal to the score threshold specified for this search
	Match and image region	Rectangles that describe the location and extent of the result instance
	Image from client transform at the center	Image from client transform at the center point of this result instance
	Diagnostics	Various diagnostic information
	PatFlex results	The PatFlex results
	PatPersp results	The PatPersp results

PatMax parameters and results summarized

RSI Search

This chapter describes RSI Search, a vision tool that lets you find patterns in images. It contains the following sections:

Some Useful Definitions on page 324 defines some terms that you will encounter as you read.

RSI Search Overview on page 324 provides an introduction to RSI Search. It describes some of the concepts that underlie RSI Search.

How RSI Search Works on page 328 provides a general description of the operation of RSI Search.

Using RSI Search on page 337 provides some guidelines for using RSI Search.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

degree of freedom: Part of a transformation that can be characterized by a single numeric value such as angle

feature: Any specific pattern of grey levels in an image. A feature is a portion of an actual image, as opposed to a model which is an idealized representation of a feature.

generalized degree of freedom: A degree of freedom other than x-translation or y-translation. For example: uniform scale, x-scale, y-scale, or rotation.

model: An idealized representation of a pattern, stored as a set of grey levels, and containing other data, such as model size and contrast.

model image: Image that contains the features you are searching for.

pose: The position and orientation of a 2D coordinate system within another 2D coordinate system. A pose comprises 6 degrees of freedom: X-translation, Y-translation, X-rotation, Y-rotation, X-scale, and Y-scale.

position: The location of a 2D point within a 2D coordinate system. A pose comprises 2 degrees of freedom: X-translation and Y-translation.

search image: Image in which to locate features similar to the model image.

search space: The collection of nominal values or ranges of values for each of the generalized degrees of freedom. A search space defines a range of possible variation between a trained model and a model instance in a run-time image.

score: Value assigned to a set of features in the search image that measures the similarity between the run-time features and the trained model. Score values are scaled to the range 0.0 to 1.0; the higher the score, the closer the match.

mask: An image used to designate each pixel in a model training image or a run-time image as *care* or *don't care*.

transformation: Mathematical representation of the equations that describe the conversion of points from one coordinate system to another coordinate system.

RSI Search Overview

The RSI Search tool is a specialized search tool that combines features of both PatMax and CNLSearch. Like those tools, RSI Search lets you train a model of the image features that you are looking for, then find instances of the trained features in a run-time image.

- Like CNLSearch, RSI Search uses *normalized correlation* search to match features from a trained model in a run-time image.
- Like PatMax, RSI Search lets you find feature instances at different rotations and scales.

Unlike other Cognex search tools, RSI Search also lets you train and locate features in RGB color images and it lets you find model instances exhibiting shear.

For most applications that require the ability to find scaled and rotated feature instances, PatMax will be faster and more accurate than RSI Search. But in a number of cases, RSI Search can solve applications that PatMax cannot:

- When using color images
- When the run-time images exhibit shear with respect to the trained model
- When using small models or model images (PatMax may be unable to train patterns from small images because they don't contain enough features)
- When run-time images contain significant texture (PatMax may run slowly if images have too many features)

RSI Search Method

You use RSI Search by training a model from a training image that contains the features of interest to your application. At run time, you supply the RSI Search tool with a run-time image that you wish to search as well as the trained model. RSI Search determines the location (or locations, since the tool can find multiple instances in a single run-time image) of features that most closely resemble the features in the model.

Like CNLSearch, RSI Search searches for and matches features by computing the correlation coefficient between the pixels in the trained model and a corresponding region of the run-time image.

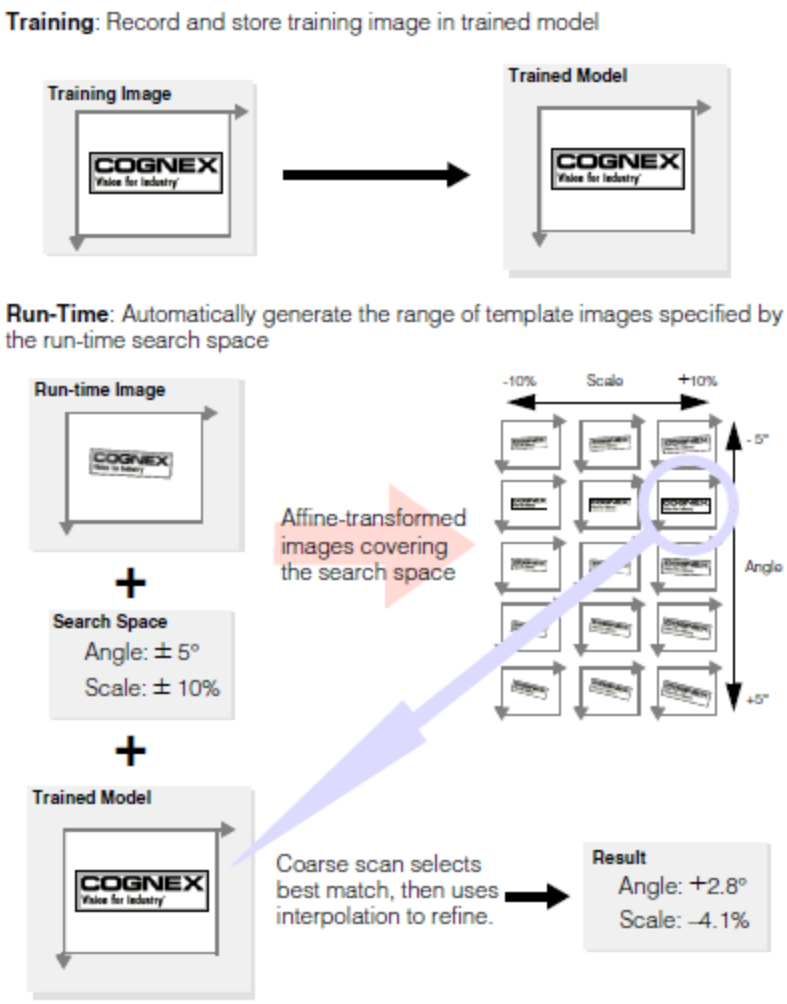
Template Generation Modes

To find features that are rotated and scaled with respect to the features used to train the model, RSI Search actually trains multiple sets of pixels that correspond to the trained features at different scales and rotations.

The tool can generate these *template images* either at training time, in which case you must specify the expected range of scale and rotation when you train the model, or at run time, in which case the tool automatically generates the template images needed, based on the range of scale and rotation that you specify for the particular search.

Run-time Template Generation (Unrestricted Mode)

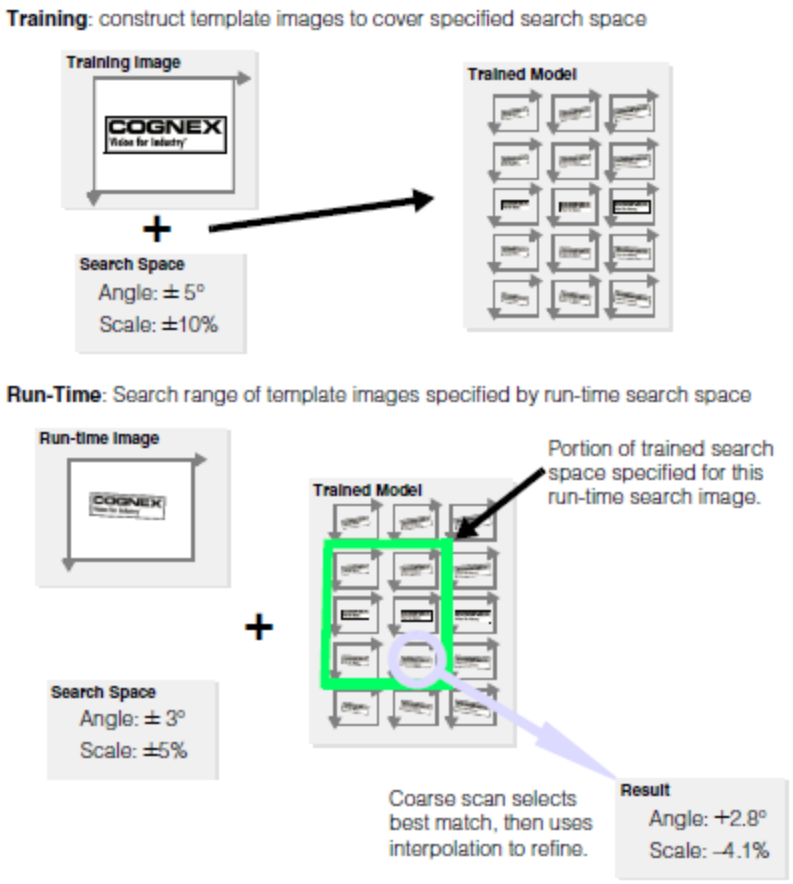
The simplest method to generate template images is to allow the RSI Search tool to automatically generate the required template images at run time, based on the run-time search space that you specify, as shown in the figure below.



Auto-generated templates

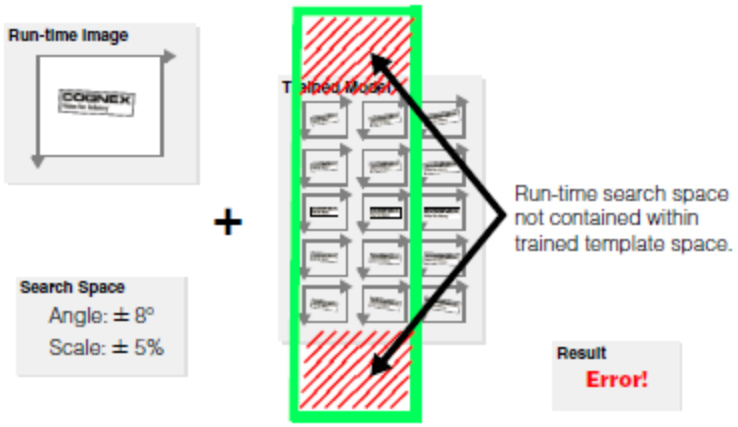
Training-Time Template Generation (Generate Templates Mode)

In cases where the range of angle and scale differences are known in advance, you can obtain faster tool execution by having the RSI Search tool generate the template images at training time. To use this mode, you must specify *at training time* the range of scale and rotation values that your application will experience at run time. RSI Search automatically creates all of the required sets of pixels to cover the search space that you specify, as shown in the figure below.



Training-time template generation

Using training-time template generation provides the fastest possible tool execution speed, but it imposes a significant constraint: You must specify, at training time, the full search space that you expect to encounter at run time. If you specify a search space at run time that is not fully enclosed within the training-time search space, the tool will generate an error and will not run. The figure below shows what would happen if the run-time search space for the trained model shown in the figure above was not fully enclosed within the trained template search space.



Invalid run-time search space

Note that even if the run-time image could be found using one of the trained templates, the tool does not run if the specified run-time search space is not contained within the trained template's search space.

The requirements for specifying the training-time search space are complex. See the section *Specifying the DOF Range with Training-Time Template Generation* on page 339 for details.

Searching

When you perform a search using RSI Search, you provide the following basic inputs:

- The image to search
- A trained RSI Search model
- The range of scale, rotation, and translation across which to search (the search space)
- The starting point or pose for the search
- The minimum number of results to find
- The acceptance and confusion thresholds for the search
- Whether or not to consider model polarity in the search
- Parameters controlling how to handle overlapping pattern instances
- Brightness and contrast thresholds for which potential instances to evaluate
- Scoring options

RSI Search uses the same strategy that the CNLSearch tool uses: instead of performing an exhaustive pixel-by-pixel search of the run-time image and template images, it performs a quick, coarse scan for likely model instances, then refines the coarse candidates using an exhaustive fine search.

RSI Search performs the coarse search across the entire search space that you specify (that is, all of the possible combinations of rotation, translation, and scale that you specify for the search). Once it has identified likely search candidates, it performs a fine search at the candidate locations.

Note: If you are generating templates at training time, then the range of scale and rotation values that you specify for the search must lie within the range of scale and rotation values that you specified when you trained the RSI Search model. The tool will not attempt to locate features with scale or rotation values for which the model is not trained. For more information, see the section *Specifying the DOF Range with Training-Time Template Generation* on page 339.

Search Score

RSI Search finds the location of a pattern in a search image based on a model image of that pattern. In addition to returning the location of the pattern in the search image, RSI Search also indicates how closely the pattern in the search image matches the pattern in the model image by returning a *score*. The score indicates how close a match exists between the trained image and the image whose location was returned. Scores range from 0.0, indicating no similarity between the model and the feature, to 1.0, indicating a perfect match.

The search score is equal to the *normalized correlation coefficient* of the pixel values in the trained model and the pixels in the corresponding region of the search image, as described in the section *Correlation Searching* on page 336.

How RSI Search Works

Like PatMax and CNLSearch, RSI Search has a training phase and a run-time phase. The successful use of the tool depends on understanding the inputs that you supply during both phases.

Training Inputs

At training time, you specify the following inputs to RSI Search:

- A training image and an optional mask
- An optional model origin
- The template generation mode
- If training-time template generation is specified, the expected range (or nominal value) for scale and rotation at run time
- If training-time template generation is specified, a flag indicating whether the generated templates consider the model origin
- Optional values for coarse and fine granularity limits
- Optional values specifying what type of compression to apply to the model
- An optional value specifying the granularity generation strategy

Each of these inputs is described in this section.

Training Image and Mask

You must supply a bound image to RSI Search when you train a model. The training image should contain the features that you expect to search for in your run-time images. In general, the smaller the training image (as long as it includes all of the features of interest), the faster RSI Search will train and run.

If your training image contains extraneous features that might not be present on all run-time images, you can exclude these features from the trained model by supplying a mask. The mask image must be the same size as the training image. Training image pixels for which the value of the corresponding mask image pixel is zero are treated as *don't care* pixels and are excluded from the trained model; training image pixels for which the value of the corresponding mask image pixel is 255 are treated as care pixels and are included in the trained model.

Model Origin

When RSI Search locates a model instance, it returns both the position and the pose of the model instance in the run time image. By default

- the model *position* is reported as the position of the origin of the model training image's client coordinate system in the run-time image's client coordinate system
- the model *pose* is reported as the pose of the model training image client coordinate system

You can specify both the simple model origin (the point within the model training image to treat as the origin) and the generalized model origin.

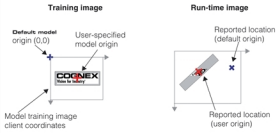
- You can specify the location within the model training image to use when reporting the pattern position.
- You can specify the pose of the model within the model training image by providing a **trainClientFromModel** transformation.

Both of these model origins are described in this section.

Simple Model Origin (Translation Only)

When you train an RSI Search model, RSI Search initializes the *model origin* to a value that you specify. When RSI Search returns the location of a model instance in a run-time image, it does so in terms of the model origin.

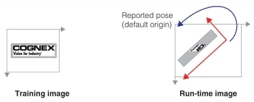
The figure below shows an example of a model being translated, scaled in the y-axis, and rotated. If you specify the default model origin in this case, the reported location may not reflect the location of the object that you are searching for.



Simple model origin

Generalized Model Origin (trainClientFromModel)

In addition to returning the position of the simple model origin in the run-time image's client coordinate system, RSI Search also returns the pose of the found model instance. By default, RSI Search returns the pose of the model image client coordinate system in the run-time image's client coordinate system, as shown in the figure below.



Reported pose

In some cases, you may want to apply a transformation to the trained model before you search for it in a run-time image. You apply such a transformation by supplying a *generalized origin*. RSI Search applies the transformation you supply to the trained model before it searches for it in the run-time image. When it returns a search result, it returns the transformation between the trained, *transformed*, model and the feature instance in the run-time image.

Reported pose for user-specified generalized origin

You typically supply a generalized origin to compensate for known scale or rotation in the model training image. If you supply a generalized origin you should keep the following points in mind:

- RSI Search will return results that describe the difference between the trained model after it has been transformed by the generalized model origin and the model in the run-time image.
- Any zone ranges and nominal values that you specify are interpreted with respect to the transformed trained model.
- Supplying a generalized origin has no effect on the speed, accuracy, or number of results produced by RSI Search.
- The x- and y-translation components of a generalized origin are equivalent to the simple (point) origin.
- If you are using training-time template generation, you can control whether or not RSI Search constructs the template images to reflect the effect of the generalized origin. This choice is discussed in section *Template Generation and Generalized Origin* on page 331.
- If you are using training-time template generation, if you wish to change the generalized origin, you must untrain the model, set the new generalized origin, then re-train the model.

Search Space (Training-Time Template Generation)

As described in the section *Template Generation Modes* on page 325, you can configure RSI Search to generate template images at training time. To do this, you must specify the *search space* for the searches that this trained model will be used with.

RSI Search supports five generalized degrees of freedom:

- Angle
- Uniform scale
- X-scale
- Y-scale
- Shear

For each of these degrees of freedom, you must specify either

- That the degree of freedom is disabled, in which case you must specify a nominal value for that degree of freedom. RSI Search will only find instances of the pattern that are close to the specified nominal value for that degree of freedom, and RSI Search will not compute a value for that degree of freedom; it will report *exactly* the nominal value that you specify for the degree of freedom.

  or

- That the degree of freedom is enabled, in which case you must specify a *zone* that defines the permitted range of values for that degree of freedom. RSI Search will find instances of the pattern that have values for the degree of freedom within the specified zone, and RSI Search will compute and report a value for the degree of freedom.

For information on how these zone and nominal values are used at run time, see the section *Specifying the DOF Range with Training-Time Template Generation* on page 339.

Uniform Scale, X-Scale, and Y-Scale

In most cases, when features vary in size, the scale variation is uniform. In some cases, however, x- and y-scale may vary independently. You can specify each of these degrees of freedom separately, and you can specify any combination of zone or nominal values.

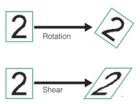
In all cases, you express scale values as a multiplier value. For example, if you specify a nominal value of 1.2 for uniform scale, then RSI Search will only find feature instances that are about 1.2 times as large as the trained features. If you enable a scale degree of freedom, you can specify the low and high limit independently. For example, you could specify a zone of 0.80 to 1.20.

Rotation and Shear

You can specify rigid rotation using the RSI Search tool. In all cases, the value that you specify is expressed in degrees. If you specify a nominal value of -10°, then RSI Search will only find features that are rotated by -10°. If you enable the rotation degree of freedom, you can specify the low and high limit independently. For example, you could specify a zone of -5° to +15°.

Shear, a rotation of the y-axis with an associated scaling of the x-axis by the inverse of the cosine of the shear angle, is also always specified in degrees.

The figure below shows the effect of shear and rotation on a trained model.



Shear and rotation

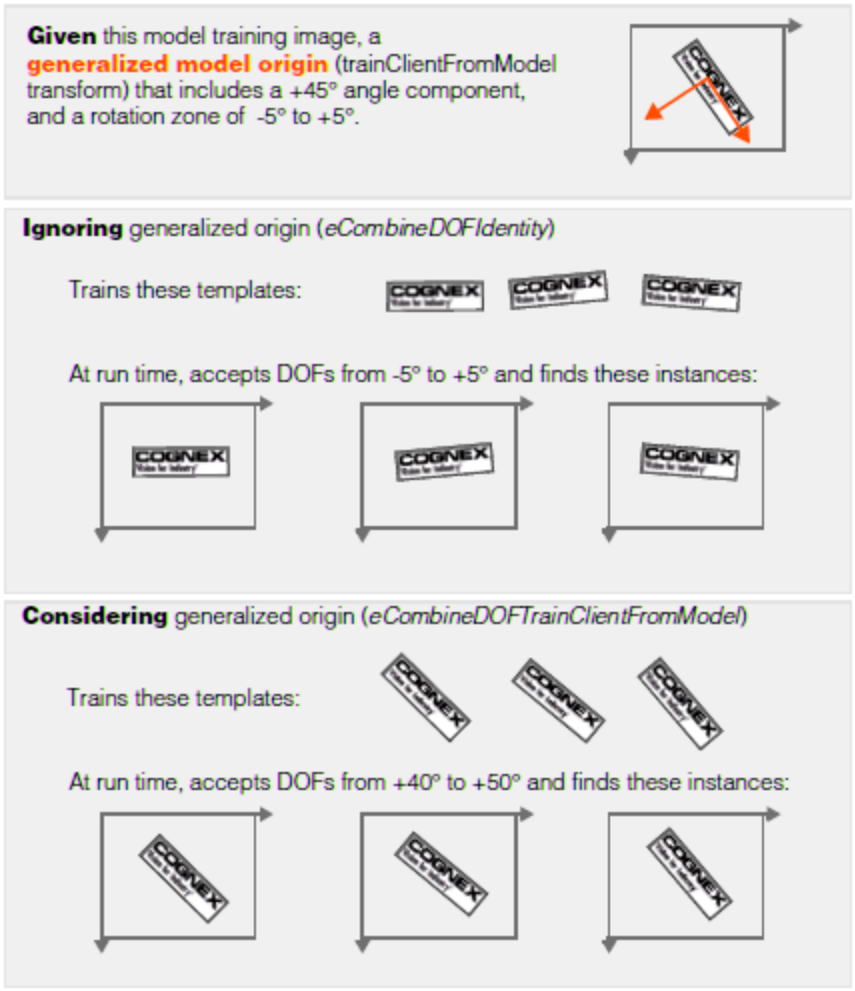
Template Generation and Generalized Origin

If you are using training-time template generation *and* you specify a non-identity generalized model origin, as described in the section *Generalized Model Origin (trainClientFromModel)* on page 330, you must specify whether or not RSI Search will combine the DOF ranges that you specify with the generalized model origin.

If you specify *ccRSIDefs::eCombineDOFIdentity*, then RSI Search presumes that the nominal or range values that you specify have not been combined with the generalized origin. If you specify *ccRSIDefs::eCombineDOFTrainClientFromModel*, then RSI Search presumes that the values have been combined with the generalized model origin.

The choice that you make determines what template images are trained and what run-time search space you must specify.

The figure below shows a simple example of the effect of this choice. In this example, the feature of interest is rotated in the model training image, and a generalized model origin is specified that corrects this rotation.

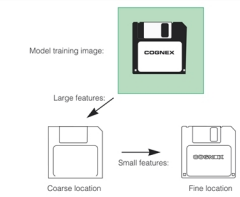


Template generation DOF combination modes

Grain Limits

RSI Search uses features of different sizes to locate similar features in run-time images. RSI Search uses large features to perform the initial coarse scan of the image, and it uses small features to precisely locate features.

For example, when RSI Search searches for a trained model of a diskette, it uses the large features from the diskette (such as the overall shape of the diskette and the outline of the label) to quickly locate the diskette, then it uses the smaller features (such as the letters on the label) to determine the precise location of the model. The figure below shows how RSI Search uses the different feature sizes.



Large features used for coarse location and small features for fine location

The features that RSI Search detects in an image are controlled by the *granularity* that it uses to analyze the image. To detect only the large features in an image, RSI Search uses a larger granularity setting. To detect the small features in an image, RSI Search uses a smaller granularity.

Granularity is expressed as the radius of interest, in pixels, within which features are detected. The figure above illustrates two important characteristics of model granularity.

- Large features such as the outline of the diskette are detected at both small and large granularity settings.
- Smaller features are present or absent from the image depending on the granularity setting.

In some cases, however, a feature might be present at a fine granularity and at a coarse granularity, but not at an intermediate granularity.

RSI Search uses a range of model granularities when it trains a model from an image; RSI Search automatically determines the optimum granularity settings when it trains a model. The smallest granularity used to detect features in the training image or shape description is called the *fine granularity limit*. The largest granularity used to detect features is called the *coarse granularity limit*.

You can override the grain limits that RSI Search determines, but in most cases the automatically computed limits provide good performance and accuracy.

Compression

RSI Search locates and scores model instances using the normalized correlation of pixel values between model and image. By default, RSI Search considers the value of every pixel in both pattern and run-time image. To improve run-time execution speed, RSI Search allows you to specify compression options at training time. When you specify a compression value, RSI Search can reduce both the amount of memory required for a trained pattern *and* the amount of time required to compute the correlation coefficient. RSI Search lets you specify the type of compression to use and how much compression to apply.

Using compression can provide a significant speed improvement, but it may cause certain marginal pattern instances to be missed. Also, the run-time scores may be less accurate when compression is enabled. You can specify, at run time, that RSI Search use an uncompressed version of the trained pattern to compute the final score.

Run Time Inputs

At run time, you specify the following inputs to RSI Search:

Run-Time Image and Mask

You must supply a bound search image when you run the RSI Search tool. The tool searches the supplied image for the trained model. If the run-time image includes features at known locations that would tend to increase image confusion or tend to prevent matches, you can exclude them by supplying a mask image.

The mask image must be the same size as the run-time image. Run-time image pixels for which the value of the corresponding mask image pixel is zero are treated as *don't care* pixels and are excluded from the search; run-time image pixels for which the value of the corresponding mask image pixel is 255 are treated as care pixels and are included in the search.

Nominal and Zone Values, Translation Uncertainty, and Start Pose

To search using RSI Search, you must specify the following information about the expected feature instances in the run-time image:

- For each of the generalized degrees of freedom (rotation, uniform scale, x-scale, and y-scale), you must either specify a nominal value or a range of values.
- You may specify a start pose within the run-time image. This specifies where in the image you expect the feature instance.
- You must specify the x- and y-axis translation uncertainty. This specifies the search area, in translation, to search.

Start Pose

You can specify an optional *start pose* for a RSI Search search. If you already know the coarse pose of the pattern instance of interest, specifying a start pose lets you reduce the time required for a RSI Search search.

The start pose is specified as a 6-degree of freedom transformation giving the pose of the pattern, as defined by the generalized by the pattern origin in the client coordinates of the run-time image.

If you specify a non-identity start pose, then RSI Search ignores any nominal values that you specify for the generalized degrees of freedom and simply computes the score at the specified pose.

If you specify a range for a degree of freedom and you provide a non-identity start pose, the range is relative to the start pose. For example, if the start pose specifies an angle of +45° and you specify a range of -5° to +5° degrees for the angle degree of freedom, RSI Search will search for pattern instances with angles of 40° through 50°.

If you specify a start pose, you can set the translation uncertainty using separate properties without needing to alter the start pose. The translation uncertainty is always specified in the units of the run-time image client coordinate system.

Note: As described in the section *Training-Time Template Generation (Generate Templates Mode)* on page 326, you configured RSI Search to generate template images at training time, then the entire range of uncertainty values must lie within the trained search space. For information on making sure that your run-time parameters conform to this requirement, see the section *Specifying the DOF Range with Training-Time Template Generation* on page 339.

Thresholds & Number to Find

When you perform a search using RSI Search, you specify parameters that RSI Search uses to determine whether a particular feature within the image is a valid instance of the model.

- The *acceptance threshold*

  The acceptance threshold is the score (between 0.0 and 1.0) that RSI Search uses to determine whether a match represents a valid instance of the model within the search image. Matches with nonzero scores greater than or equal to the acceptance threshold are valid matches.

  You use the acceptance threshold to indicate to RSI Search the degree of image degradation that you expect in search images. If you expect search images to be degraded, you should specify a lower acceptance threshold.

- The number of instances of the model you expect the search image to contain

  RSI Search returns the location of every instance of the model within the search image that has a score that exceeds the acceptance threshold, up to the number of instances you specify. If there are fewer instances of the model with scores above the acceptance threshold than you specify, RSI Search might return the location and score of some additional instances with scores below the acceptance threshold, although those instances will be marked as not found.

- The *confusion threshold*

  The confusion threshold is the score (between 0.0 and 1.0) that represents the highest score that a feature that is *not* an actual instance of the model will receive. You should always set the confusion threshold greater than or equal to the acceptance threshold.

  RSI Search treats the confusion threshold you specify as a hint about the nature of the image being searched. You should specify a high confusion threshold to indicate that the scene contains features that resemble the model but are not valid instances of the model; specify a low confusion threshold to indicate that the only features in the scene that resemble the model are valid instances of the model.

  You use the confusion threshold to indicate the difficulty you expect RSI Search to have discriminating between valid instances of the model and other features in the image. If you expect RSI Search to have difficulty discriminating which are valid instances of the model, you should specify a higher confusion threshold.

In general, a higher confusion threshold can increase the reliability of searches at the cost of somewhat slower searching. A properly selected confusion threshold lets RSI Search provide the best balance of reliability and speed.

Polarity

RSI Search evaluates the similarity of feature matches in the run-time image to the trained model by computing the normalized correlation coefficient of the pixel values in the two images, as described in the section *Correlation Searching* on page 336. This correlation value is returned as the *search score*. Search scores can range in value from -1.0 through +1.0. A score of +1.0 indicates a perfect match, while a score of -1.0 indicates a perfect mis-match. A score of 0.0 indicates no match. By default, RSI search clamps search scores to the range 0.0 through 1.0. All mis-match results receive a score of 0.0.

You can specify that RSI Search ignore the polarity of search scores. In this case, negative scores are mapped to positive scores.

Note: RSI Search, polarity refers to inverting the pixel values while for PatMax, polarity refers to the polarity of image features (dark-to-light or light-to-dark). The RSI Search tool can not find model instances which exhibit nonlinear intensity differences from the trained model, regardless of whether polarity is ignored or considered. Only PatMax and CNLSearch (in nonlinear mode) can find such model instances.

Brightness and Contrast Thresholds

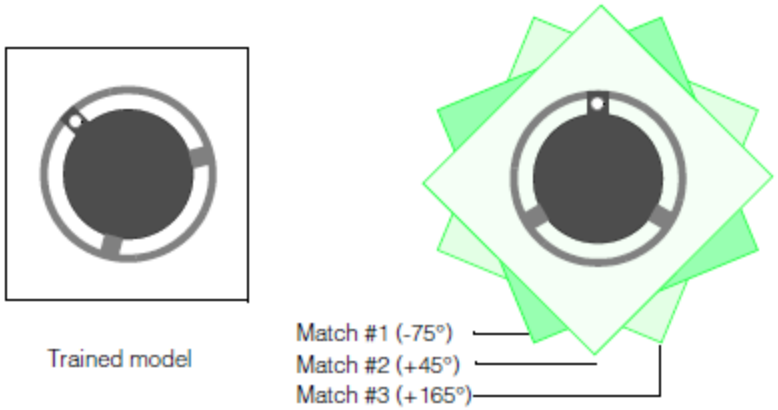
RSI Search performs a basic histogram analysis of both the training image and the matched pixels in the run-time image. When you run RSI Search, you can limit search results to those with a specified brightness and contrast variation from the trained image.

The brightness variation threshold is expressed as a low and high multiplier value for the mean of the training image pixel values. The contrast variation threshold is expressed as a low and high multiplier value for the standard deviation of the trained image pixel values.

Using these thresholds can allow you to reduce the acceptance threshold without finding spurious matches.

Model Instance Overlap

By default, when RSI Search finds multiple feature instances in the run-time image that largely overlap each other, it assumes that these instances actually represent the same features in the image. For example, if you use RSI Search to search for the part shown in the figure below with the angle degree of freedom enabled, RSI Search will locate three instances of the part at the same location but at three different angles. By default, RSI Search discards all but the strongest match and returns a single result.



RSI Search selects the best of overlapped instances

For some applications, it might be necessary or desirable to obtain a separate result for each of the matching instances shown the figure above.

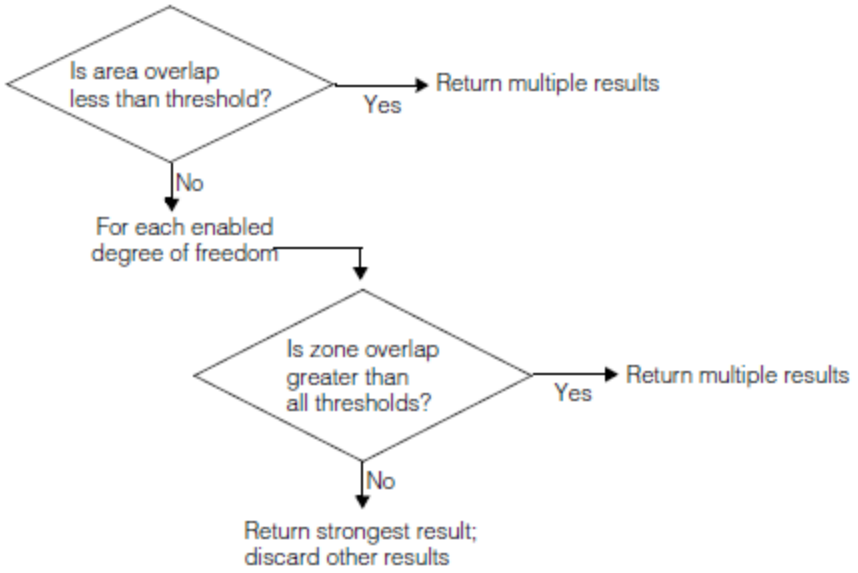
RSI Search lets you control how multiple overlapping model instances are handled by letting you specify an *overlap threshold*. You can specify an area overlap threshold (expressed as the percentage of the model training area that overlaps) and a separate zone overlap threshold for each enabled degree of freedom.

Note:
The area overlap and zone overlap thresholds behave differently.

As you increase the area overlap threshold, RSI Search requires that multiple instances overlap to a greater degree before it treats them as a single instance. As you decrease the area overlap threshold, RSI Search requires that multiple instances overlap to a lesser degree before it treats them as a single instance.

As you increase a zone overlap threshold, RSI Search requires a smaller difference in the degree of freedom value between multiple instances before it treats them as a single instance. As you decrease a zone overlap threshold, RSI Search requires a greater difference in the degree of freedom value between multiple instances before it treats them as a single instance.

RSI Search uses the procedure shown in the figure below to determine whether or not to discard overlapped model instances.



Overlap threshold processing

Scoring Controls

If you specified that RSI Search use compression when you trained the model, as described in the section *Compression on page 333*, you can configure the tool to compute the final match score using an uncompressed version of the trained model.

Specifying uncompressed scoring results in slightly slower execution speed, and it may result in the tool returning different model instances than are returned when scoring is performed using the compressed model.

How RSI Search Finds Features

This section describes the details of how RSI Search works at run time.

Correlation Searching

RSI Search determines the degree to which features in a run-time image are similar to the features in a training image by computing the correlation coefficient of the two sets of pixel values. The correlation coefficient is expressed as a number

between -1.0 and 1.0. A correlation coefficient of 1.0 means that the pixel values in the two images are perfectly matched. A correlation coefficient of -1.0 means that the pixel values in the two images are perfectly mismatched. A correlation coefficient of 0 means that pixel values in the two images are randomly different.

The figure below shows three sets of image pairs, one pair with a positive correlation (a), one pair with a negative correlation (b), and one pair with an insignificant correlation (c).

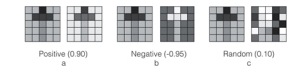


Image pairs showing different correlation coefficients

Mathematically, the correlation coefficient r of a model and a corresponding portion of an image at image offset (u,v) is given by $$r\left(u,v\right)=\frac{\left[N\sum_i I_iM_i-\left(\sum_i I_i\right)\left(\sum_i M_i\right)\right]}{\sqrt{\left[N\sum_i I_i^2-\left(\sum_i I_i\right)^2\right]\left[N\sum_i M_i^2-\left(\sum_i M_i\right)^2\right]}}$$

where

N is the total number of pixels.

I_i is the value of the image pixel at $(u+x_i, v+y_i)$.

M_i is the value of the corresponding model pixel at the relative offset (x_i, y_i).

The value of r is always in the range -1.0 to 1.0, inclusive. A value of 1.0 signifies a *perfect match* between the area of the image and the model. Specifically, if r = 1.0, there exist some values a and b such that for all i: $I_i = aM_i + b, a > 0$

Sub-Pixel Accuracy

RSI Search determines the sub-pixel location at which the normalized correlation coefficient is the highest between the model pixels and the run-time image pixels. You can select the method that the tool uses to compute this sub-pixel location:

- *Standard* mode uses a basic mathematical model for estimating the sub-pixel position with the highest score.
- *High-Accuracy* mode computes the sub-pixel location of the correlation score peak using a mathematical model that better characterizes how the normalized correlation coefficient changes at small sub-pixel image offsets. (Standard mode uses a mathematical model that may not accurately model changes in the correlation score at small image offsets.)

Color Search

When you use RSI Search with color images, the tool searches all three planes of the color image simultaneously, and it computes the search score across all the pixels in all three planes.

You can alter the relative weight that the tool applies to the individual planes in computing the score.

Note: You can only search RGB color images using RSI Search. Other color spaces, such as HSI, are not supported. If you wish to use HSI images, you must convert them to RGB.

Using RSI Search

This section provides some guidelines for using RSI Search.

Training a Model

This section discusses in more detail the operations that RSI Search performs to train a model. It describes the parameters that you supply when you specify a model.

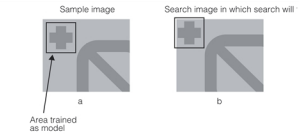
Selecting the Model Image

Once you have identified the feature within the model training image that you want to use for a model, you need to define the rectangular region of the image that you actually train as a model.

RSI Search only finds instances of the model that are entirely contained within the search image and that are no closer than two pixels to the edge of the search image. RSI Search does not find model instances that are only partially contained within the search image, and it does not find model instances that extend into a 2-pixel-wide boundary around the outside edge of the image.

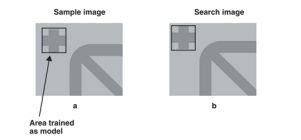
Remember that RSI Search considers the entire area that you specify to be part of the model.

The figure below shows an image where an otherwise acceptable model image is trained incorrectly, resulting in search failures. The cross-shaped fiducial in the first sample image (a) is used as the model feature. Because the area used to define the model is so close to the edge of the image, when the search image is shifted (b), the model area falls partially outside the image and RSI Search fails to locate the model.



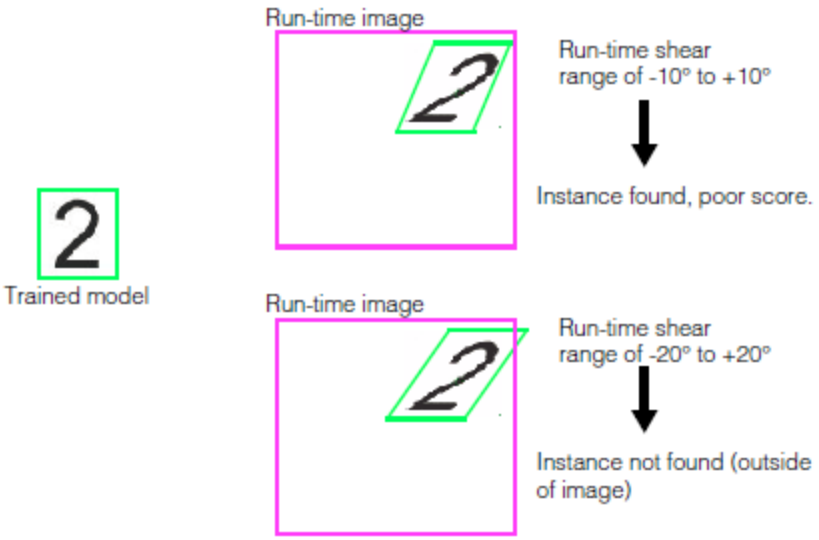
Poorly trained model

The image below shows how the same feature in the same image can be trained correctly. By making sure that sufficient area exists between the edge of the model and the edge of the image, you ensure that the search will succeed even when presented with images in which the position of the feature has moved.



Properly trained model.

In some cases, specifying a larger value for the Shear degree of freedom can cause a search to fail if the sheared model area overlaps the edge of the search image, even though the value is closer to the actual amount of shear in the instance. The shear and rotation figure in the section Rotation and Shear shows this effect.



Expanding range of shear DOF can cause search failures

Specifying the DOF Range with Training-Time Template Generation

If you use training-time template generation, as described in the section *Training-Time Template Generation (Generate Templates Mode)* on page 326, RSI Search will not create *any* affine-transformed versions of the search image at run time. When you specify the degrees of freedom at training time, you must make sure that RSI Search trains all the required images to handle the expected range of run-time images.

The general requirements for specifying the training-time degrees of freedom are:

- The search space that you specify at run time for RSI Search must be entirely contained within the search space that you specified at training time. If, at run time, you specify a search space that does not meet this requirement, the tool will not run.
- The actual set of affine-transformed images created during training must contain a set of pixels that correspond to the actual pixels in the run time image. If this requirement is not met, the tool may not generate an error, but it will fail to find the pattern.

While the first requirement may seem straightforward, you need to consider the following factors when computing whether or not your training time search space will, in fact, contain your run-time search space:

- Any difference between the client coordinates of the training image and the client coordinates of the run-time image. The training search space must be expanded to reflect the difference, in any degree of freedom, between these two coordinate systems.
- The use of a non-identity generalized model origin. As discussed in the section *Template Generation and Generalized Origin* on page 331, you can control how the model origin is combined with the training-time degrees of freedom.
- The use of a non-identity start pose. If you specify a start pose, then the run-time degrees of freedom are interpreted *relative to that start pose*, as discussed in the section *Start Pose* on page 334

The effect of the first and third of these factors is not known by RSI Search until you actually run the search; it is your responsibility, at training time, to compute the effect of these factors on the ultimate run-time search space and set the training time search space appropriately.

The training time search space must contain the ultimate search space as determined by the start pose, any difference between the training and run-time images' client coordinates, any non-identity run-time start pose, and the run-time range of degrees of freedom. As described in the section *Training-Time Template Generation (Generate Templates Mode)* on page 326, when you train an RSI Search model, the tool trains all of the rotated and scaled versions of the training image required to cover the search space that you specify. This section provides detailed information about how that search space is determined.

In general, the search space that you specified when you trained the RSI Search model must encompass the entire search space that you specify when you search using that model. For example, if you specify a scale range of 0.8 to 1.2 and a rotation of -5° to +5° and then specify a nominal scale of 2.0 or a rotation range of 0° to +10° at run time, the tool will throw an error.

There are two special circumstances under which you must adjust the training-time search space limits differently than this.

- If the client coordinate systems of the training image and the run-time image have different values for scale or rotation, then you must adjust the training-time search space to reflect the difference between the coordinate systems.

  For example, if you are working with a sub-sampled run-time image but a full-resolution training image, then you must adjust the uniform scale degree at training time to account for the scale difference between the two images.

  In nearly all cases, however, the client coordinate systems will be the same for training and run-time images.

- If, at run time, you will supply a start pose that has a nonzero rotation or a scale other than 1.0, then you must adjust the training-time search space to reflect the difference between the coordinate systems.

Start Pose and Search Space

When you RSI Search, you must supply a start pose that defines where to start searching for the trained model in the run-time image. The combination of the start pose and run-time degrees of freedom must lie within the trained range for the degrees of freedom.

The search space that you specify must meet the following requirements:

- If you specify a nominal value for scale or rotation, then this value must, with respect to the scale and rotation component of the search space, lie entirely within the range of scale and rotation values for which the model was trained. If the model was trained with a nominal scale or rotation value, then the nominal scale or rotation must be the same as the nominal scale or rotation.

  For example, if the model was trained with a rotation range of -5° to +15°, and a start pose rotation of +5° degrees was specified, then a nominal or zone value in the range -10° to +10° would be valid.

- At some point within the search space that you specify, the trained model must be fully enclosed within the search image (excluding a two-pixel boundary at the edge of the search image. It is not a requirement that the trained model be fully enclosed within the search image at all points within the search space, nor that it be fully enclosed at the start pose, only that at some point within the search space that the model be fully contained within the search image.

Searching for the Model Within the Search Image

This section discusses in more detail the operations that RSI Search performs when it searches for an instance of the model image within the search image.

Search Area

When you supply a search image to RSI Search, it searches the entire area of the image except for a 2-pixel-wide border around the outside edge of the image.

Confusion and Acceptance Thresholds

For recommendations on selecting confusion and acceptance thresholds, see the section *Selecting a Confusion Threshold and an Acceptance Threshold* on page 268 of the CNLSearch chapter.

Search Results

RSI Search returns a collection of information called a *search result* for each instance of the model that it finds in the search image, up to the number of instances you specify for the search. The results are returned in score order, with the highest score first.

Shape Finder Tool

This chapter describes the Shape Finder tool, a tool that finds geometric shapes in an image.

This section and *Some Useful Definitions on page 341* give an overview of the chapter material and define some terms you will encounter as you read.

The chapter has the following major sections:

Shape Finder Tool Overview on page 341 provides an overview of finding geometric shapes in images.

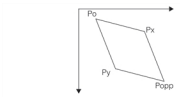
How The Shape Finder Tool Works on page 343 provides a description of how the Shape Finder tool accomplishes its task.

Using the Shape Finder Tool on page 352 describes how to use the Shape Finder tool in an application.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

affine rectangle: A quadrilateral shape where the opposite sides are parallel. Defined by the class, **ccAffineRectangle**. See the following figure:



Caliper tool: A CVL vision tool for finding edge points in images. A caliper is a window defined by an affine rectangle where the Caliper tool searches for an edge point. The Shape Finder tool uses the Caliper tool to find edge points in images.

edge point: A point found by the Caliper tool where there is a significant grey scale difference in an image.

Fitting tool: A CVL vision tool that fits a specified shape to a set of edge points. The Shape Finder tool uses the Fitting tool to fit lines, circles, and ellipses to sets of points.

pose: A transform from one coordinate system to another that allows six degrees of freedom; x and y translation, x and y scale, x and y rotation.

start pose: The transform used to map a search model into a vision tool work space. For example, mapping an expected circle into the client coordinate space of the image passed to the Circle Finder tool.

Shape Finder Tool Overview

The Shape Finder tool finds geometric shapes in an image. When you run the tool the run-time parameters specify which geometric shape the tool finds. Currently, run-time parameters are provided for a line segment, circle, and ellipse. When used for these specific applications, the tool is sometimes called the Line Finder tool, Circle Finder tool, or Ellipse Finder tool. This overview discusses Shape Finder tool operation that applies to these geometric shapes. *How The Shape Finder Tool Works* on page 343 covers each shape individually.

To find these geometric shapes the Shape Finder tool uses the Caliper tool to find shape edges and specific Fitter tools to fit the found edge points to the geometric shapes. The Shape Finder tool is provided to create an easy to use interface for performing a specific function. You can of course, write your own shape finder by calling the Caliper tool and Fitter tools directly from your application. However, this will require more work on your part. If you wish to read about these underlying tools see *Caliper Tool* on page 189 and *Fitting Tool* on page 42.

The Shape Finder tool runs in client coordinate space. You specify shapes and calipers in their own space and provide the tool with a start pose that maps these shapes into client coordinates. Results are also in client coordinates.

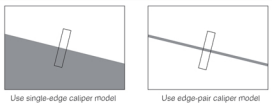
Run Mode

The Shape Finder tool can be run in *manual* mode or *automatic* mode. In manual mode you specify each caliper, where it is placed, and its Caliper tool run-time parameters. This provides you with a great deal of control for unusual applications. In automatic mode you specify the number of calipers and one Caliper tool run-time parameter set to be used with all the calipers. You then specify a region on the shape where you want the calipers placed and the tool places them automatically. This is the easiest way to use the tool.

The Caliper Tool

The Shape Finder tool finds edge points on a shape by repeatedly calling the Caliper tool. An important part of using the Shape Finder tool is selecting the right number of calipers and placing them optimally in the image. Calipers must be chosen so that the shape edge you wish to find is the only edge inside the caliper. Other edges may confuse the Caliper tool and cause it to fail.

When you run the Shape Finder tool you provide the Caliper tool run-time parameters it will use. For this reason, you must be familiar with the Caliper tool and how to use it in your application. For example, you would run the Caliper tool in single-edge mode to find the left-hand example line in the figure below, while the right-hand example line has two edges and requires that you run the Caliper tool in edge-pair mode. Set the Caliper tool run-time parameters so that the tool finds a result point that the Shape Finder tool can use as an input to a Fitter tool.



Caliper examples

Also, the Caliper tool *scan mode* can be useful when working with circles and ellipses. In this mode the tool scans the image at a number of different incremental angles to find the caliper angle that produces the best contrast. See *Scan Mode* on page 195.

Fitter Tools

Fitter tools include the Line Fitter tool, Circle Fitter tool, and the Ellipse Fitter tool. The tool used depends on the run-time parameters you provide to the Shape Finder tool. Once the Shape Finder tool has found the edge points using calls to the Caliper tool, it passes these points to the proper fitter tool to fit the shape. An important part of this procedure is that fitter tools require a minimum number of points. The Line Fitter tool requires at least two points, the Circle Fitter tool requires at least three points, and Ellipse Fitter tool requires at least five points. For better results, always provide more than the minimum number of required points.

Ignoring Points

Fitter tools have a run-time parameter that specifies a number of input points it can ignore to make a better fit. This allows the tool to ignore outlier points and just use the best-fit points to fit the shape. When you specify the number of calipers to use in your application allow extra calipers and then specify a number of points that can be ignored by the fitter. In most cases, this will produce more accurate results.

In some cases a caliper may fail and not produce a point to be carried forward to the Fitting tool. In this case, the Fitting tool will have fewer points to work with than your design calls for. Also, if you have specified that the Fitting tool can ignore some points, caliper failures can leave the Fitting tool with fewer points than needed for a successful fit. To protect against this condition, the Shape Finder tool has a feature you can enable that will decrement the Finder tool ignore count for every caliper that fails. When you use an ignore count with your Fitting tool, we recommend you also enable this auto-decrement feature.

Results

When you call the Shape Finder tool you provide a result object, either a line result, circle result, or an ellipse result. The result contains the following information:

- If the shape was found. A shape is found if its computed error is less than the error *threshold* specified in the Fitting tool parameters. For example, the **ccCaliperCircleFinderAutoRunParams** include a **ccCircleFitParams** object that contains an error *threshold* value.
- The found shape.
- The position of the found shape.
- The Fitter tool results object.
- The tool execution time.

If the result indicates the shape was not found, it is possible that it was actually found but its computed error exceeded the error threshold. If this happens, be aware that you can retrieve the out of tolerance shape from the result object and investigate it if you wish.

Performance

When you call the Shape Finder tool you provide a run-time parameters object and a results object. For applications where you repeatedly run the tool on many images, you can improve your performance by reusing the same run-time parameters object and results object with each call. When you do this the tool uses previously cached parameters and saves setup time.

When you run the Shape Finder tool in automatic mode, you must provide a copy of the shape you expect to find that is close to the actual shape. The closer your expected copy is to the actual shape, the more accurate your result will be. To improve accuracy, run the tool and obtain the result. Then use the found shape in the result as your expected shape and run the tool again. Your second result will be more accurate.

How The Shape Finder Tool Works

When you run the Shape Finder tool you provide a run-time parameters object and a results object. These objects determine which shape the tool will find. Objects are currently provided for a line segment, circle, and ellipse. Each of these shapes is described separately in the following sections.

Finding Line Segments

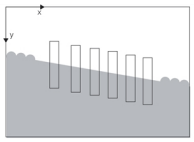
This section describes how the Shape Finder tool finds a line segment in an image.

Caliper Placement and the Expected Line

To use this tool you must know the approximate location of the line you wish to find in the image. You then specify and place calipers so that the calipers are guaranteed to find the line edges in the image. You can place calipers at specific locations (*manual* placement), or you can specify an expected line along which the tool places calipers (*automatic* placement). The expected line can be defined as two end points, or by using an affine rectangle. The following sections describe each of these options.

Manual Caliper Placement

When you use manual caliper placement you specify a number of affine rectangles, each representing a caliper window. You also specify a corresponding set of caliper run-time parameters to be used when the Caliper tool is called. Using this approach, each caliper can be uniquely specified. *Example of manual caliper placement* on page 349 shows an example of six unique calipers defined to find the line in the image.



Example of manual caliper placement

All affine rectangles have the same orientation. See the **ccAffineRectangle** reference page for more detailed information.

For this example, the Caliper tool is called six times using a different Caliper tool run-time parameters object for each call.

Automatic Caliper Placement

To use automatic caliper placement you define an *expected line* and a number of calipers. The tool then distributes the calipers along the expected line. The expected line you specify should be close to the actual line you wish to find in the image.

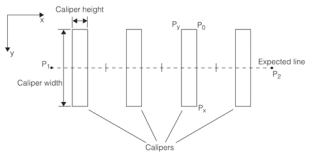
In automatic mode, you specify the geometry of one caliper, the number of calipers you wish to use, and a set of run-time parameters. The tool decides where to place the calipers, and aside from this difference, each caliper run is identical.

The two methods of defining an expected line are discussed int he following sections.

Defining a Line with Endpoints

The line you define should be close to the actual image line you wish to find. If you plan to run the tool on many images and you expect some variations from one image to the next, make your expected line sufficiently shorter than the actual line so that the calipers will always find a line edge.

The Line Finder tool divides the expected line into equal segments, one per caliper, according to the number of calipers you specify. It then centers a caliper on each line segment. An example showing four calipers is shown in the figure below.

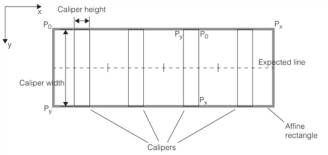


Expected line example using endpoints

Defining a Line Using an Affine Rectangle

With this method also, the line you define should be close to the actual image line you wish to find. If you plan to run the tool on many images and you expect some variations from one image to the next, make your expected line sufficiently shorter than the actual line so that the calipers will always find a line edge.

The expected line is the line that divides the affine rectangle exactly in half. See the figure below.



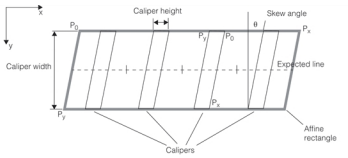
Expected line example using an affine rectangle

P_0 in the figure above refers to the upper left-hand corner of the affine rectangle. See the **ccAffineRectangle** reference page for more detailed information.

Once you define the expected line using the affine rectangle, placement of the calipers along the expected line is the same as that described in the previous section. Note here that the affine rectangle width defines the caliper width, and the rectangle length defines the expected line length. You need only specify the caliper height. All calipers have the same dimensions.

Adding Skew

Normally calipers are perpendicular to the expected line as shown in *Expected line example using an affine rectangle* on page 344. However, it is important to make sure that the only edge inside each caliper is the line edge you wish to find. To avoid extraneous edges inside calipers you may wish to skew the calipers by specifying a skewed affine rectangle. See the example in the figure below.

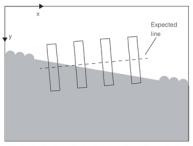


Skewed calipers

Running the Tool

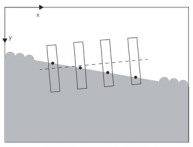
Once the calipers are placed, the tool runs each caliper to find an edge point. The tool assumes that the line is the only edge point inside the caliper so be sure you design your application so that no other features can be mistaken for a line edge.

The figure below is an example of an expected line with four calipers. The example purposely uses an expected line angle somewhat different from the actual line angle to illustrate the angle tolerance you have to work with. The critical factors include making sure the image line edge falls inside each caliper, and ensuring no other features inside the calipers might be mistaken for a line edge.



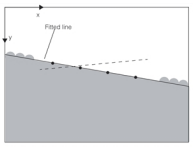
Example, expected line with calipers

The figure below shows the edge points found after the tool runs the calipers.



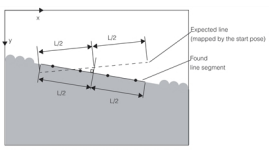
Example, found edges

The found edge points are then applied to the Line Fitter tool which fits an infinite length line to the points. The fitted line is shown in the figure below.



Fitted line example

The infinite length line shown in the figure above is shortened to a line segment using a projection of the expected line, and this shortened line segment is then returned as the found line. The figure below shows how the expected line length is used to establish the found line segment.



Returned line segment example

Line Finder Results

The following information is available with the Line Finder tool results:

found
A boolean indicating whether the Line Finder tool found the line.

fitterResultsValid
A boolean indicating whether the Line Fitter tool had a valid result.

pose
A transform that maps the found line segment to the expected line, in client coordinates.

caliperResults
A vector of caliper tool results, one result for each caliper run.

edgePositions
A vector of edge points found by the Caliper tool, one for each caliper run.

caliperTime
The time in seconds used by the Caliper tool for all calipers run.

totalTime
The time in seconds used by the Line Finder tool. (Includes the **caliperTime**).

lineFit
The Line Fitter tool result.

line
The found line (an infinite length line).

lineSeg
The found line segment (the same length as the expected line).

lineFitTime
The time in seconds used by the Line Fitter tool.

Finding Circles

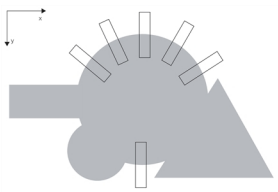
This section describes how the Shape Finder tool finds a circle in an image.

Caliper Placement and the Expected Circle

To use this tool you must know the approximate location of the circle you wish to find in the image. You then specify and place calipers so that the calipers are guaranteed to find the circle edges in the image. You can place calipers at specific locations (*manual* placement), or you can specify an expected circle around which the tool places calipers (*automatic* placement). The following sections describe each of these options.

Manual Caliper Placement

When you use manual caliper placement you specify a number of affine rectangles, each representing a caliper window. You also specify a corresponding set of caliper run-time parameters to be used when the Caliper tool is called. Using this approach, each caliper can be uniquely specified. The figure below shows an example of six unique calipers defined to find the circle in the image.



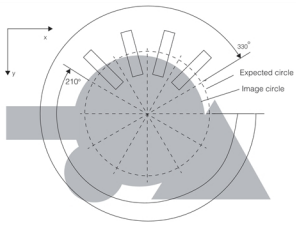
Example of manual caliper placement

For this example, the Caliper tool is called six times using a different Caliper tool run-time parameters object for each call.

Automatic Caliper Placement

In automatic mode, you must know the approximate location of the circle you wish to find in the image. You specify this circle in circle space, and include a start pose that locates the expected circle in client coordinate space. You then choose a number of calipers and an angular range where the tool places calipers along the circle perimeter. You need not place calipers around the complete circle circumference, however you do need to find at least three circle edge points for the tool to fit a circle. Choose an angle range where you are sure there are no other image edges besides the circle, and choose three or more calipers for this angular region. The tool will divide the angular range into equal size segments, one segment for each caliper, and place the calipers in the center of each segment along the arc.

The figure below shows an example of an expected circle and four calipers. The angle range specified for this example is 210° - 330°.

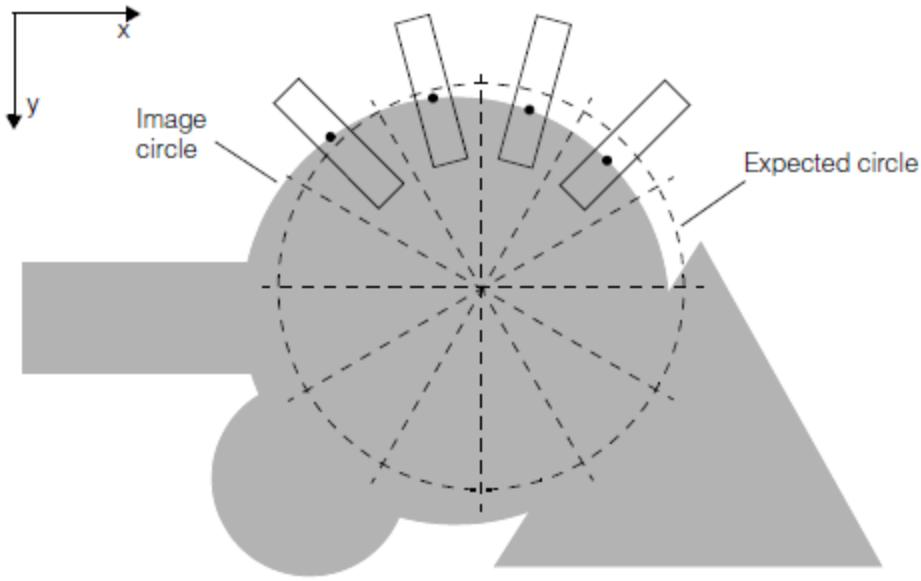


Example of expected circle and four calipers

Note that although the expected circle is not exactly coincident with the image circle, it is close enough so that the calipers all see an image circle edge. This is the important consideration.

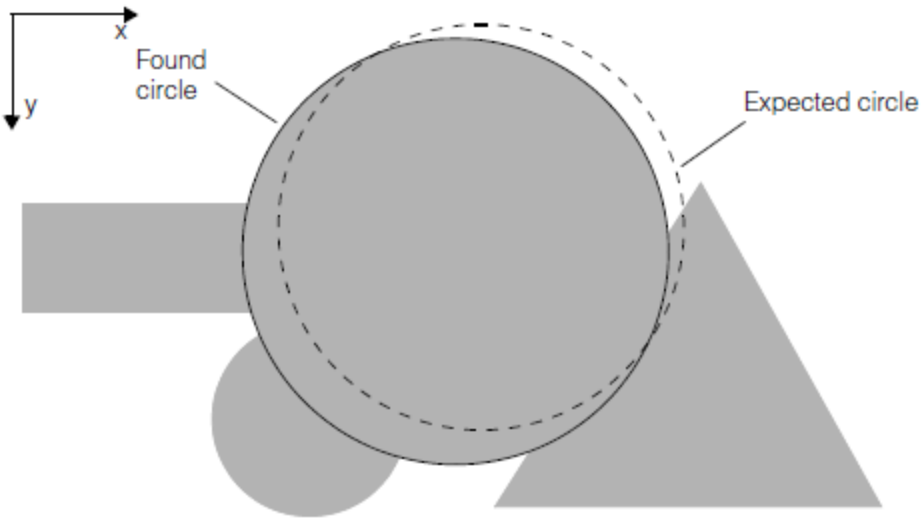
Running the Tool

After placing the calipers, the tool calls the Caliper tool once for each caliper, using the same run-time parameters for each call. The figure below shows the edge points found by the Caliper tool.



Example, found edges

The found circle edges are applied to the Circle Fitter tool which fits a circle to the points. The figure below shows the final fitted circle which is included in the tool result.



Example, found circle

Circle Finder Results

The following information is available with the Circle Finder tool results:

found
A boolean indicating whether the Circle Finder tool found the circle.

fitterResultsValid
A boolean indicating whether the Circle Fitter tool had a valid result.

pose
A transform that maps the found circle to the expected circle, in client coordinates.

caliperResults
A vector of caliper tool results, one result for each caliper run.

edgePositions
A vector of edge points found by the Caliper tool, one for each caliper run.

caliperTime
The time in seconds used by the Caliper tool for all calipers run.

totalTime
The time in seconds used by the Circle Finder tool. (Includes the **caliperTime**).

circleFit
The Circle Fitter tool result.

circle
The found circle.

position
The location of the found circle center.

radius
The found circle radius.

circleFitTime
The time in seconds used by the Circle Fitter tool.

Finding Ellipses

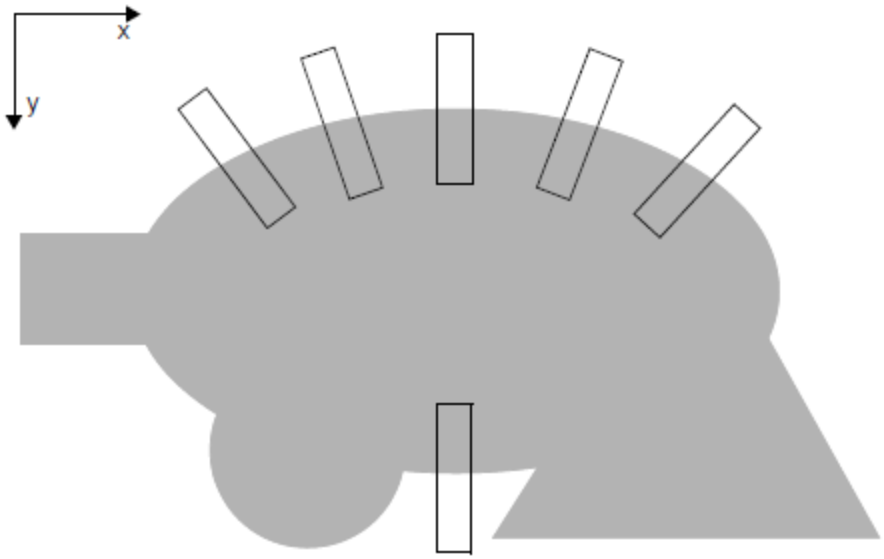
This section describes how the Shape Finder tool finds an ellipse in an image.

Caliper Placement and the Expected Ellipse

To use this tool you must know the approximate location of the ellipse you wish to find in the image. You then specify and place calipers so that the calipers are guaranteed to find ellipse edges in the image. You can place calipers at specific locations (*manual* placement), or you can specify an expected ellipse around which the tool places calipers (*automatic* placement). The following sections describe each of these options.

Manual Caliper Placement

When you use manual caliper placement you specify a number of affine rectangles, each representing a caliper window. You also specify a corresponding set of caliper run-time parameters to be used when the Caliper tool is called. Using this approach, each caliper can be uniquely specified. The figure below shows an example of six unique calipers defined to find the ellipse in the image.



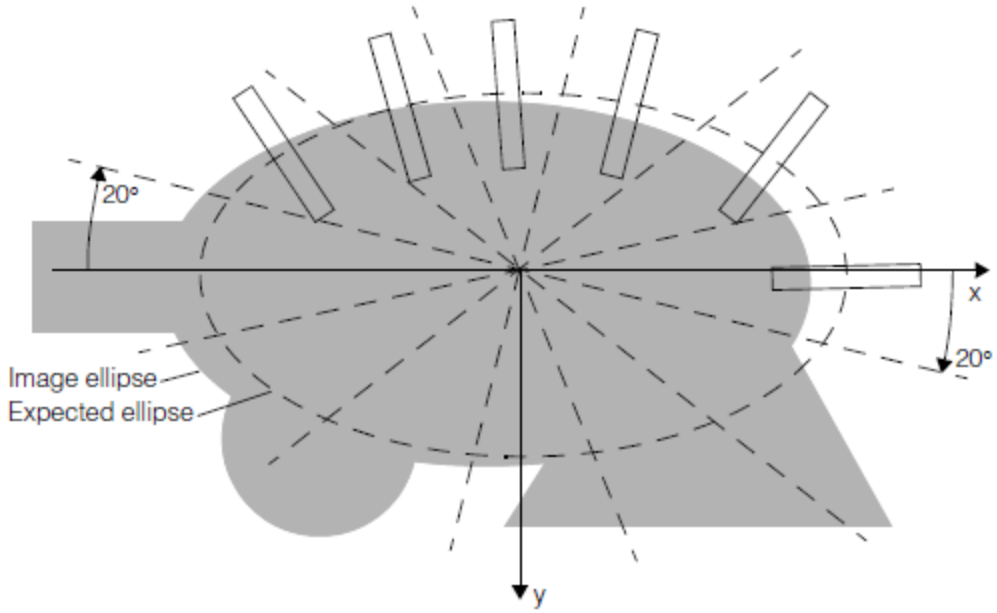
Example of manual caliper placement

You specify the calipers in affine rectangle space and provide a start pose that locates the calipers in client coordinate space. For this example, the Caliper tool is called six times using a different Caliper tool run-time parameters object for each call.

Automatic Caliper Placement

In automatic mode, you must know the approximate location of the ellipse you wish to find in the image. You specify this ellipse in ellipse space, and include a start pose that locates the expected ellipse in client coordinate space. You then choose a number of calipers and an angular range where the tool places calipers at approximately equal distances along the ellipse perimeter. You need not place calipers around the complete ellipse circumference, however you do need to find at least five ellipse edge points for the tool to fit an ellipse. Choose an angle range where you are sure there are no other image edges besides the ellipse, and choose three or more calipers for this angular region. The tool will divide the angular range into approximately equal size segments, one segment for each caliper, and place the calipers in the center of each segment along the ellipse arc, orthogonal to the arc tangent at each point.

The figure below shows an example of an expected ellipse and six calipers. The angle range specified for this example is 200° - 380°.



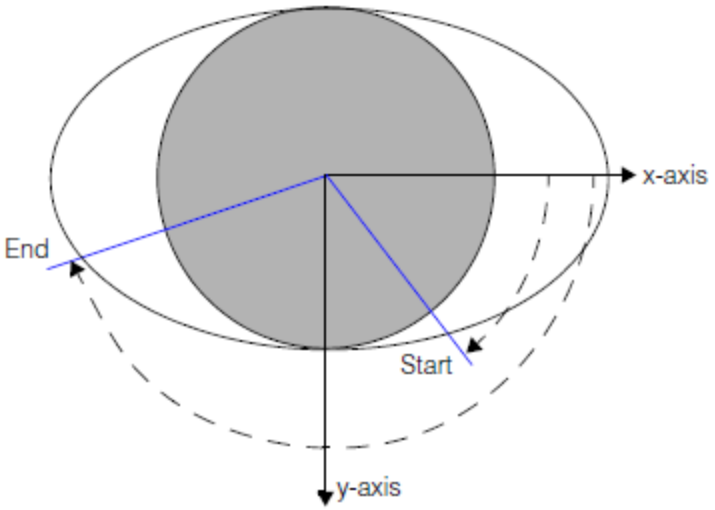
Example of expected ellipse and six calipers

Note that although the expected ellipse is not exactly coincident with the image ellipse, it is close enough so that the calipers all see an image ellipse edge. This is the important consideration.

Choosing the Angle Range

To specify the angle range where calipers will be automatically placed, you specify the starting angle of the range, and the ending angle. These angles are measured with respect to the ellipse segment projected onto a unit circle. Angles are measured from the x-axis of the unit circle.

The starting angle is simply the angle from the x-axis to the circle radius that marks the beginning of the range (Start). Likewise, the ending angle is the angle from the x-axis to the radius that marks the end of the range (End). See the figure below.



Ellipse angle range and the unit circle

The **ccEllipse** object that represents the expected ellipse includes a **ccEllipseUnit** structure (see the **ccEllipse** reference page). This structure defines the transformation from ellipse coordinates to unit circle coordinates. You can use this

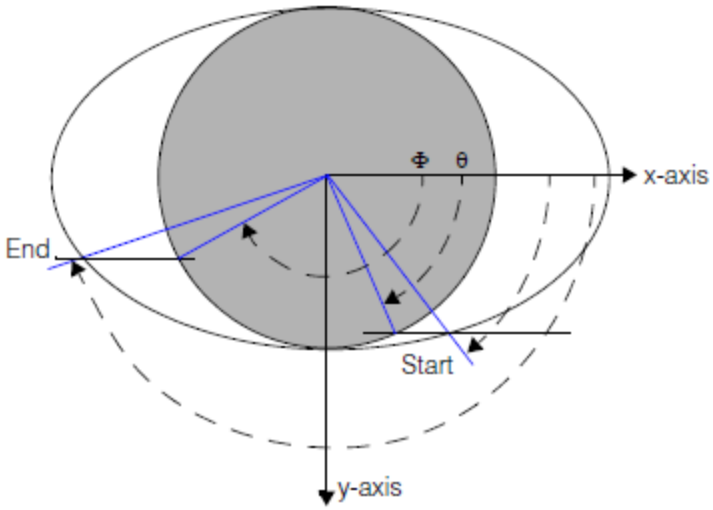
transformation to calculate the starting and ending angle values. For a given point *p* on the expected ellipse, its angle *phi* in the unit coordinate system is computed by the following expression:

```
ccEllipse expectedEllipse;
cc2Vect center; // default constructor creates as (0,0)
ccPoint p;

(expectedEllipse.unit().U_ * (p - center)).angle();
```

Apply the above expression in turn to each endpoint *p* of the expected elliptical arc to find the starting and ending angles of the range. Use these unit circle angles to find an ellipse with the Shape Finder tool.

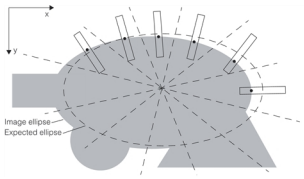
If you do not have a **ccEllipse** object to start from, you can obtain approximate start and end angles graphically using the projection method illustrated in the figure below. Draw a unit reference circle whose radius is one-half of the minor axis of the expected ellipse; this reference circle fits just inside the ellipse as shown in the figure below. Draw a line parallel to the major ellipse axis, and passing through the point where the Start vector intersects the ellipse. The point where this line intersects the reference circle is the projection of the ellipse onto the circle. Draw a new circle radius to this point and measure the angle θ. In the same manner, project the ending ellipse point onto the unit circle and measure the angle Φ. The start angle θ and the end angle Φ specify the angle range you should use to find an ellipse with the Shape Finder tool.



Projection method of finding angles

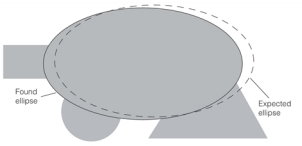
Running the Tool

After placing the calipers, the tool calls the Caliper tool once for each caliper, using the same run-time parameters for each call. The figure below shows the edge points found by the Caliper tool.



Example, found edges

The found ellipse edges are applied to the Ellipse Fitter tool which fits an ellipse to the points. The figure below shows the final fitted ellipse (Found ellipse) which is included in the tool result.



Found ellipse

Ellipse Finder Results

The following information is available with the Ellipse Finder tool results:

found
A boolean indicating whether the Ellipse Finder tool found the ellipse.

fitterResultsValid
A boolean indicating whether the Ellipse Fitter tool had a valid result.

pose
A transform that maps the found ellipse to the expected ellipse, in client coordinates.

caliperResults
A vector of caliper tool results, one result for each caliper run.

edgePositions
A vector of edge points found by the Caliper tool, one for each caliper run.

caliperTime
The time in seconds used by the Caliper tool for all calipers run.

totalTime
The time in seconds used by the Ellipse Finder tool. (Includes the **caliperTime**).

ellipseFit
The Ellipse Fitter tool result.

ellipse
The found ellipse.

position
The location of the found ellipse.

ellipseFitTime
The time in seconds used by the Ellipse Fitter tool.

Using the Shape Finder Tool

You program the Shape Finder tool by calling the global function **cfCaliperFindShape()** and passing it an image, a run-time parameters object, and a results object. The image contains the shape you wish to find, and the run-time parameters object provides all the information about how to find the shape. The results object is used by the Shape Finder tool to store the results.

The run-time parameters object and the results object are each unique for the geometric shapes the tool handles (line, circle, and ellipse). For example, if you wish to find a circle in an image, you must pass **cfCaliperFindShape()** a circle run-time parameters object and a circle results object.

You can also run the tool by calling one of the shape-specific global functions **cfCaliperFindLine()**, **cfCaliperFindCircle()**, or **cfCaliperFindEllipse()** and passing the same arguments. These functions are redundant, however, you may wish to use them instead of **cfCaliperFindShape()** to add clarity to you code.

The table below summarizes the classes and global functions you will use when you program the Shape Finder tool.

	Line	Circle	Ellipse
Global function that runs the tool	**cfCaliperFindShape()**	**cfCaliperFindShape()**	**cfCaliperFindShape()**
Redundant global function	**cfCaliperFindLine()**	**cfCaliperFindCircle()**	**cfCaliperFindEllipse()**
Run-time parameters (manual mode)	**ccCaliperLineFinder ManualRunParams**	**ccCaliperCircleFinder ManualRunParams**	**ccCaliperEllipseFinder ManualRunParams**
Run-time parameters (automatic mode)	**ccCaliperLineFinder AutoRunParams**	**ccCaliperCircleFinder AutoRunParams**	**ccCaliperEllipseFinder AutoRunParams**
Results	**ccCaliperLineFinder Result**	**ccCaliperCircleFinder Result**	**ccCaliperEllipseFinder Result**

Shape Finder tool class summary

Note: Default constructed run-time parameters objects are not valid for running the tool. You must set parameters such as *numCalipers* and *caliperSize* to valid values.

Run Mode

You run the Shape Finder tool either in manual mode or automatic mode. This choice refers to how you use the Caliper tool. In manual mode you specify each caliper and each Caliper tool run-time parameter set, and the Shape Finder tool calls Caliper once for each caliper you specify. Each caliper can be unique regarding its size, shape, location, and how its run.

In automatic mode you specify just one caliper and one caliper run-time parameter set. You tell the Shape Finder tool approximately where to place the calipers and how many to use, and the tool does the rest. Automatic mode requires less work for you but you give up some low-level control of the Caliper tool.

The mode you use is specified by the run-time object type you pass in when you call **cfCaliperFindShape()**. For example, when you run the circle finder in manual mode you pass the tool a **ccCaliperCircleFinderManualRunParams** object, and to use automatic mode you pass it a **ccCaliperCircleFinderAutoRunParams** object. The similarities and differences between these two run-time objects are compared and discussed in the next section.

Run-time Parameters

The Shape Finder tool run-time parameters objects are complex and contain other objects that specify how you wish to run the tool. The following table compares the manual mode and automatic mode run-time objects for a circle, and includes comments you may find helpful in writing your application.

Manual mode ccCaliperCircleFinder ManualRunParams members	Automatic mode ccCaliperCircleFinder AutoRunParams members	Comments
ccCircleexpectedCircle()	**ccCircleexpectedCircle()**	The circle you expect to find in images.
vector<**ccAffineSamplingParams**> affineSamplingParamsList()		Specifies calipers for the Caliper tool.
vector<**ccCaliperRunParams**> caliperRunParamsList()	**ccCaliperRunParams caliperRunParams()**	Caliper tool run-time parameters.
	c_Int16 numCalipers()	The number of calipers to use.
	ccDPair caliperSize()	The size of each caliper. (all the same size)

Manual mode ccCaliperCircleFinder ManualRunParams members	Automatic mode ccCaliperCircleFinder AutoRunParams members	Comments
	double caliperWidth()	The caliper width. (all the same size)
	double caliperHeight()	The caliper height. (all the same size)
	bool centrifugal()	Direction of the caliper search.
	ccDPair caliperSampling()	Sampling rate for the calipers
	ccAngleRange angleRange()	Caliper location
	Interpolation interpolationMethod()	Caliper sampling method.
	bool placeCalipers Symmetrically()	Automatic caliper placement method.
c_Int16 minNumCalipers()	**c_Int16 minNumCalipers()**	The minimum number of calipers required.
ccCircleFitParams circleFitParams()	**ccCircleFitParams circleFitParams(()**	The run-time parameters to be passed to the Fitting tool.
cc2Xform startPose()	**cc2Xform startPose()**	The start pose for the expected circle.
bool descrmentNumIgnore()	**bool descrmentNumIgnore()**	See *Ignoring Points* on page 342.

Circle run-time parameters

Notice that except for the Caliper tool parameters all other members are the same for both modes. Since you are providing the run-time parameters for the Caliper tool it is imperative that you know how to program the Caliper tool. The same applies to the Fitting tool. If you have questions about these tools see *Caliper Tool* on page 189 and *Fitting Tool* on page 42, and the associated class reference pages.

Results

When you call the Shape Finder tool you pass it an empty results object and the tool fills it in with results. The Shape Finder tool calls the Caliper tool and the Fitting tool as part of its algorithm. Results from these two tools then become part of the Shape Finder tool result.

Each supported shape has its own results class as follows:

Shape	Class
Line	**ccCaliperLineFinderResult**
Circle	**ccCaliperCircleFinderResult**
Ellipse	**ccCaliperEllipseFinderResult**

The table below summarizes the contents of an example circle results object (type **ccCaliperCircleFinderResult**) showing the members and comments you may find useful in your applications. The table breaks out results carried forward from the Caliper and Fitting tools as well as results unique to the Shape Finder tool.

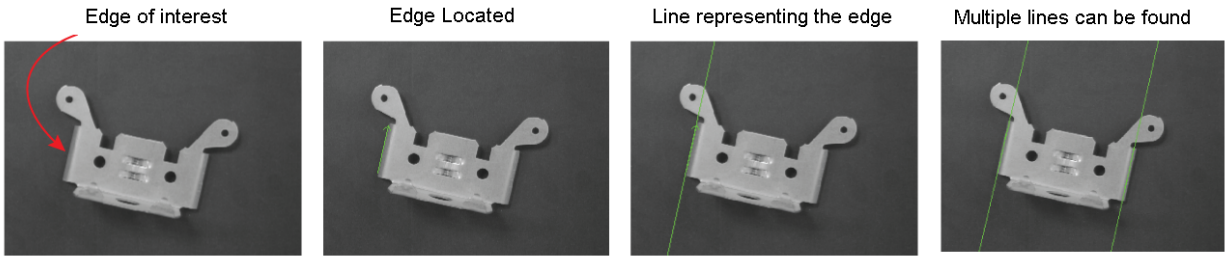
Member	Shape tool result	Caliper tool result	Fitting tool result
bool found()	Indicates if the tool found a valid shape.		

Member	Shape tool result	Caliper tool result	Fitting tool result
bool fitterResultsValid()			Did the Fitting tool have a valid result.
cc2Xform pose()	Locates the found shape relative to the expected shape.		
vector<**ccCaliper ResultSet**> **caliperResults()**		Results from each call to the Caliper tool.	
vector<**cc2Vect**> **edgePositions()**		Found edges from the Caliper tool.	
double caliperTime()		Caliper tool run time.	
double totalTime()	Run time for all the tools.		
ccCircleFitResults circleFit()			The fitting tool result object.
ccCircle circle()	The found circle		
cc2Vect position()	The position of the origin of the found circle.		
double radius()	The radius of the found circle.		
double circleFitTime			The Fitting tool run time.

Circle Finder tool results

LineMax Tool

The LineMax tool locates one or more line segments corresponding to linear edge points in your acquired images:



A LineMax tool locates probable edge points and fits the best possible line segment based on criteria that you specify. You configure the tool with the expected orientation of the segment you want to find, its polarity (dark to light transitions), as well as other line characteristics.

By default a LineMax tool locates a single line segment, but can be configured to locate multiple line segments when the correct CVL license is present. See the CVL User's Guide for details on CVL licenses.

The LineMax tool is more robust than the line finding capabilities of the Shape Finder tool which requires precisely placed calipers in order to find individual edge points. Cognex recommends the LineMax tool for new applications, and continues to support the line finding options of the Shape Finder tool for existing applications.

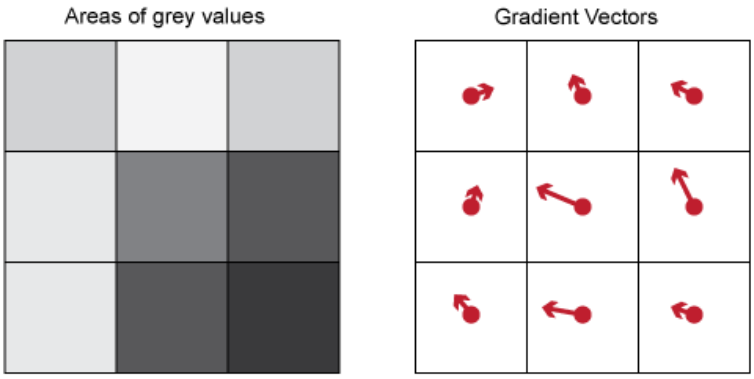
Find the LineMax API at *%VISION_ROOT%\defs\ch_cvl\linemax*.h.

Gradient Vectors

A LineMax tool analyzes a runtime image in order to generate a set of gradient vectors representing all the edge information it contains. Each gradient vector has a direction and a magnitude:

- The direction points to where the image increases in brightness.
- The magnitude indicates the strength of the contrast between this area of the image and the surrounding areas.

The following figure shows an example set of gradient vectors generated from image information:



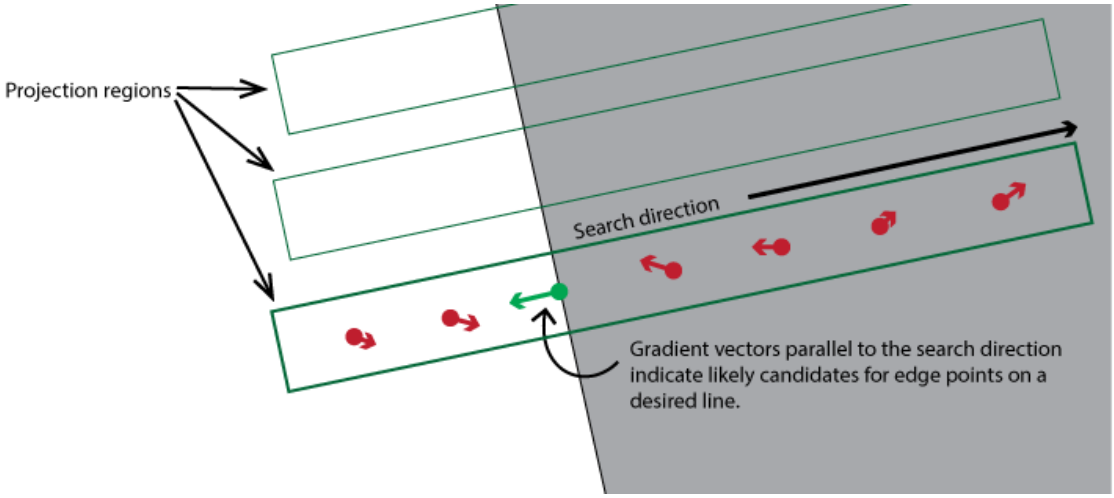
The set of all the gradient vectors generated from a runtime image represents a *gradient field*. The LineMax tool defines potential edge candidates and performs line fitting on the gradient field and not the original runtime image.

Edge Point Detection

A LineMax tool uses a set of parameters to extract edge point candidates for the features you want to detect. Setting the best values for these parameters allows a LineMax tool to find the desired edges in the shortest execution time possible.

- To reduce the amount of data to process and improve execution time, the tool samples the runtime image and generates gradient vectors for square areas of the image defined by a gradient kernel with a **gradient kernel size** of 2:
  - Runtime images with sharp edges might require a small value for accuracy.
  - Runtime images with soft features might require larger values for edge location.

  In general, smaller values lead to results with higher accuracy but higher execution times, while larger values can reduce execution time at the cost of some accuracy.
- The tool performs *gradient field projection* to analyze the gradient field with a series of projection regions along a specified search direction perpendicular to the orientation of the expected line segment:



- The parameter **projection length** defines the size of the projection region and ultimately the quantity of regions used to analyze each gradient field. A small value includes more gradient vectors than a projection region with a larger value. In general a small value requires more time to execute while a larger value can improve the execution speed of the tool but might not detect the edges you want the tool to locate.
- Each projection region generates multiple edge points to pass along to the line fitting phase.
- Gradient vectors must have a magnitude higher than an **absolute contrast threshold** and a **normalized contrast threshold**.

Raise the value of **normalized contrast threshold** to discard false edge points in bright areas of your image. Be aware that setting it to 1 discards all edge points, making the tool effectively nonfunctional.

Line Fitting

A LineMax tool uses a set of line-fitting parameters for fitting all edge points to one or more line segments.

Inliers and Outliers

The LineMax tool defines found edge points as either *inliers* or *outliers*:

- The tool identifies an edge point as an *inlier* if it is included in the fit for one of the line results.
- Edge points not capture by any line results are identifies as *outliers*.

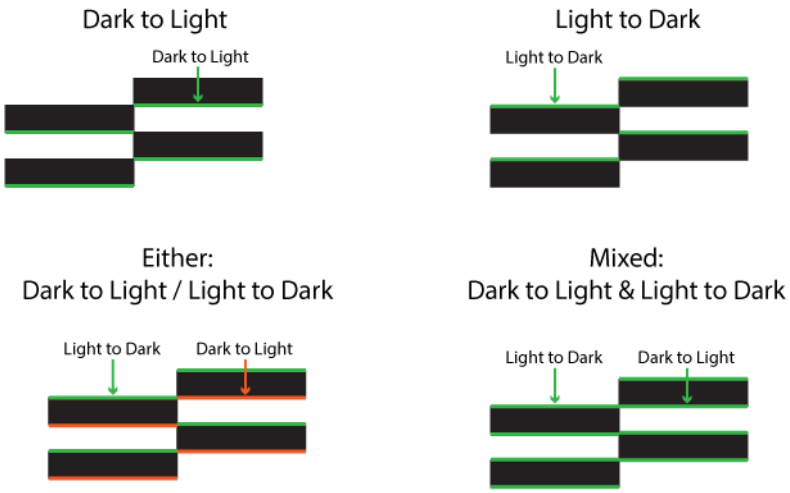
A LineMax tool can be configured to generate information about all edge results in the current image without finding any lines by setting the number of line segments to locate to zero.

Edge Constraints

The tool relies on a number of limitations regarding found edges points to determine if they fit along a desired line.

Edge Polarity

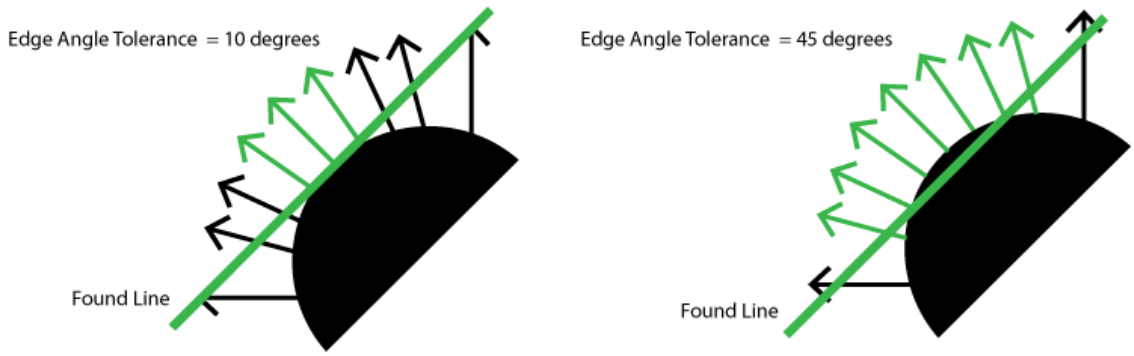
You must specify the polarity of edge points that fit into the line segment you want the tool to locate. The following figure illustrates all possible polarities:



Edge Angle Tolerance

The LineMax tool uses an edge angle tolerance to limit the difference between the gradient vector direction for an edge point and the orientation of the normal to the line segment. Candidate gradient vectors with a direction within that range are included as inliers to the found line segment.

As you increase the edge angle tolerance, you allow the tool to consider more edge points as inliers and change the location of the found line, as illustrated in the following figure:



Edge Distance Tolerance

A LineMax tool fits a line to candidate edge points with respect to a distance tolerance, measured in client units of the image.

As you increase the edge distance tolerance, you allow the tool to consider more edge points as inliers and change the location of the found line, as illustrated in the following figure:

In general, set a distance tolerance threshold in client units that is equivalent to a distance of 1-3 pixels.

Fitting Algorithm

The tool supports two line-fitting algorithms:

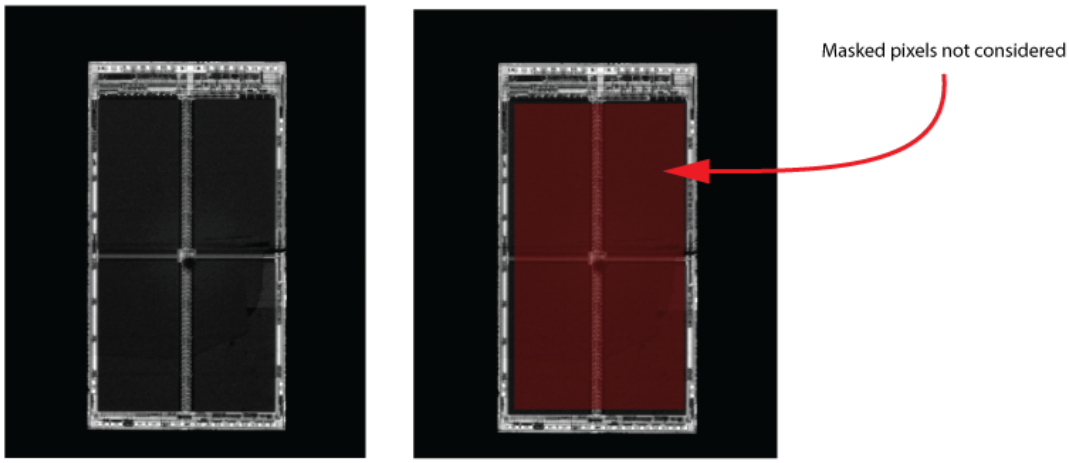
- The Random Sample Consensus algorithm (RANSAC), an algorithm for robust fitting of edge points in the presence of many edge point candidates.
- Exhaustive mode: All edge point combinations are evaluated for line fitting.

Cognex recommends you use the default RANSAC mode and switch to Exhaustive mode only if necessary, as this mode can require more execution time.

Image Masking

The LineMax tool supports the use of adding a mask image to prevent the tool from searching for edge points in selected sections of the image.

For example, the following figure shows an input image with no mask and another where all internal features are masked out (in red) from consideration:



Results

The tool returns the following information about found line segments:

- The (x,y) location, in coordinates of the selected space, for the start and end of the fitted line segment
- The number of inliers and the coverage score, which represents the percentage of projection regions that contributed to this found line segment.
- Its polarity, intensity and amount of contrast
- A residual RMS fit error, relative to the client space of the input image, from the contributing inliers

In addition, the tool can be configured to return no line segments but still generate a set of edge results available through the LineMax API.

Auto-Select Tool

This chapter describes the Auto-select tool, a tool that automatically selects part of an image to use for model training.

This section and *Some Useful Definitions on page 361* give an overview of the chapter and define some terms you will encounter as you read.

Auto-Select Tool Overview on page 361 describes the basic capabilities of the Auto-select tool.

How the Auto-Select Tool Works on page 362 provides information that helps you make the most effective use of the Auto-select tool.

Using the Auto-Select Tool on page 369 outlines the steps to using the Auto-select tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

Auto-select tool: Name of the tool when performing search operations

enhanced mode: Enhanced mode of operation of the Auto-select tool. Allows you to perform search or query operations (but not both). Uses modified scoring metrics, provides support for masking, and allows you to set overlap constraints. See also non-enhanced mode.

masking: Eliminates certain portions of an image from the analysis. This is useful, for example, for ignoring areas with known defects. Masking is supported only when the Auto-select tool is run in enhanced mode.

model: Representation of the pattern or feature you are searching for

Model Advisor: Name of the tool when performing query operations

model training image: Image that contains the pattern or feature you are searching for; used to train a model

model training window: Portion of a larger image to train as a model

non-enhanced mode: Original mode of operation of the Auto-select tool. This is essentially a compatibility mode, in which the tool produces the same results as in versions prior to CVL 6.0.1. See also enhanced mode.

overlap: An amount by which model areas can overlap and still be considered two separate windows. If models overlap by more than this amount, they are considered to be the same model, and only one of them will be returned from a search operation. You can set overlap constraints only when the Auto-select tool is run in enhanced mode.

query operation: When performing a query operation, you provide an explicit set of candidate window locations to evaluate. The tool evaluates and scores these windows using the same metrics as during a search operation. Query operations are supported only when the Auto-select tool is run in enhanced mode. When running query operations in this mode, the tool is known as the Model Advisor. See also search operation.

search operation: When performing a search operation, the Auto-select tool evaluates all possible candidate window locations. Search operations are supported in both enhanced and non-enhanced modes. When running search operations in either non-enhanced or enhanced mode, the tool is known as the Auto-select tool. See also query operation.

Auto-Select Tool Overview

Pattern-location tools require that you supply a *model training image*. The pattern-location tool extracts information from the model training image and then uses that information to find instances of the model in one or more search images.

The Auto-select tool takes an input image, a model size, and information about which pattern-location tool you are using, and returns one or more suggested model training windows within the image. You can then use the returned window to train a model. The Auto-select tool supports both PatMax and CNLSearch.

The Auto-select tool works exclusively in image coordinates, ignoring any client coordinate transforms of images and masks passed to it. The tool returns recommended training windows with locations specified in image coordinates.

What Makes a Good Model Training Image

Proper model selection is one of the most important factors in obtaining good results from any pattern-location tool. Effective model training images tend to share the following general characteristics:

- Uniqueness

  Model training images should not be confusing with respect to other features in a typical search image.

- Balanced horizontal and vertical elements

  Most pattern-location technologies work better when searching for patterns with a balance of strong horizontal and vertical elements.

- Redundancy

  Models that can still be found even when parts of the model are occluded are more effective.

It can be difficult for a human operator to select an optimal model training window with all of these characteristics from an input image. This is one area where machine vision can play an important role.

What the Auto-Select Tool Does

The Auto-select tool scans an input image and calculates scores for metrics that indicate the degree to which the location would make a good model training window. You can specify the size of the training window.

You supply an input image (often a typical search image) and a set of search parameters for a particular pattern-location tool, such as CNLSearch or PatMax.

When running in non-enhanced mode, the Auto-select tool evaluates all possible locations of the input image for certain preselected metrics. If you supplied a 512x512 pixel input image and a 64x64 pixel model size, the Auto-select tool would need to consider more than 200,000 possible locations. Since few models depend on 1- or 2-pixel wide features, you can speed up the process of auto-selection by specifying a sub-sampling factor. Sub-sampling lets the Auto-select tool exclude features below a certain size from consideration when evaluating model window locations.

If you specify a sub-sampling factor, the Auto-select tool will construct a reduced-resolution version of the input image before computing the scores. This reduces the amount of time required for the tool to run, but it can also reduce the quality of the tool's model training window selection.

The Auto-select tool reduces the resolution of the input image by the sub-sampling factor you specify. If you specify a factor of 4, then the Auto-select tool reduces the resolution of a 512x512 input image to 128x128, and evaluates 16x16 pixel model training windows. This reduces the number of possible locations to 12,544, greatly reducing the amount of time required to produce a result.

When running in enhanced mode, the Auto-select tool allows you to specify which locations to evaluate and which metrics to calculate for those locations. The tool may run faster in this mode. See *Modes of Operation* on page 363 for details.

How the Auto-Select Tool Works

This section describes in detail how the Auto-select tool works. The information in this section can help you get the best results from the Auto-select tool.

Modes of Operation

The Auto-select tool has two modes of operation: *non-enhanced mode* and *enhanced* mode. You specify which mode to use when constructing the run parameters object (see *Specifying the Mode* on page 369).

The non-enhanced mode is the legacy mode of operation. In this mode, the tool calculates scores for three metrics in a cascading manner for all possible window locations. Results of search operations in non-enhanced mode are identical to those produced in versions prior to CVL 6.0.1. See the next section, *Score Values* on page 363, for information on how scores are calculated.

The enhanced mode is a more flexible mode of operation. It allows you to perform either query operations on specific locations or search operations on the entire image. When performing query operations in enhanced mode, the tool is also known as the Model Advisor. As query operations have only a limited set of locations to evaluate, they may produce faster results. See also *Enhanced Mode* on page 365 for further information.

Score Values

When the Auto-select tool runs on the input image, it can compute the following scores for each location under evaluation:

1. The *symmetry score*, which is a measure of the symmetry of the model window in relation to the symmetries of nearby windows.
2. The *orthogonality score*, which is a measure of the degree to which the model window contains strong vertical and horizontal edges and the presence of an equal balance of vertical and horizontal edges.
3. The *uniqueness score*, which is a measure of the uniqueness of the model window with respect to the rest of the input image, as measured by the pattern-location tool you specify.

In non-enhanced mode, the Auto-select tool always computes a symmetry score for every possible location in the input image. It computes an orthogonality score only for locations that receive a high symmetry score, and a uniqueness score only for locations whose symmetry and orthogonality scores are both high. In enhanced mode, the search operation behaves similarly, but the query operation computes all of the scores you specify.

Each of these scores is discussed in more detail below.

Symmetry Score

The symmetry score is based on two factors:

1. The degree of symmetry of the model training window about a pair of axes.
2. The degree to which the model training window is more symmetrical than nearby windows.

Note: The second factor applies only to non-enhanced mode. In enhanced mode, the symmetry score is not affected by the symmetry of nearby windows.

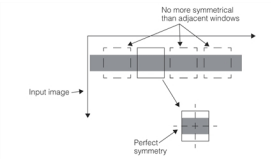
The symmetry score considers the overall pattern of pixel values when it assesses the symmetry of a region.

The figure below shows some examples of more and less symmetrical model images.

Symmetry

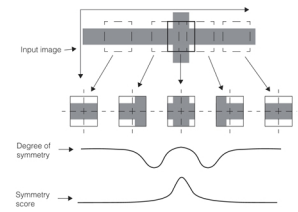
Note: Keep in mind that the symmetry *score* reflects the symmetry of the window *and* the degree to which the window is more symmetrical than other nearby windows (in non-enhanced mode only).

The figure below shows an example of a window within an input image that would receive a low symmetry score, even though the pattern of pixel values within the window is perfectly symmetrical. Because other windows in the immediate vicinity are equally symmetrical, the window receives a low symmetry score.



High symmetry with low symmetry score

The figure below shows an example of where a model training window with a high degree of symmetry receives a high symmetry score. Even though the center window is no more symmetrical than the windows at either side of the image, because the immediately adjacent windows are less symmetrical, the window receives a higher symmetry score.



Symmetry peak

Orthogonality Score

The orthogonality score provides a measure of whether or not the model contains a balance of strong vertical and horizontal edge features. A low orthogonality score can indicate an overall absence of strong vertical and horizontal features *or* the lack of a balance between strong vertical and horizontal features.

The figure below shows examples of model patterns that would receive low and high orthogonality scores.

Low and high orthogonality patterns

Uniqueness Score

The uniqueness score is computed by actually training the model training window, then searching the entire input image for the trained model, using the pattern-location tool and parameters that you specify. The score that the trained region receives is compared with the highest score received by another region. The greater the difference between the two scores, the higher the uniqueness score.

For more information on what affects the uniqueness score, you should consult the documentation for the pattern-location tool you are using (PatMax or CNLSearch).

Training and Runtime Parameters

The Auto-select tool takes as inputs the runtime parameters for the pattern location tool you specify. The tool uses the parameters you supply to perform the test searches. You should make sure that you specify parameters that are similar to the parameters you intend to use to perform actual searches.

The Auto-select tool does not allow you to specify the training-time parameters used for the test searches. For PatMax, the tool uses the following default training-time parameters:

- The pattern is trained for the algorithm specified by the runtime parameters.
- The pattern uses the automatically selected granularity limits.
- The pattern polarity must match (ignorePolarity is false).

For CNLSearch, the tool uses for its training-time parameters the values for algorithm and edge hysteresis thresholds that you specify in the runtime parameters.

Combining Score Values

In non-enhanced mode, once the tool has computed the three score values, it combines the scores to produce an overall score for each location. You control how the scores are combined using the following two parameters:

- The weight to apply to each component score when computing the overall score
- The method of computing the overall score (geometric mean or arithmetic mean)

Enhanced Mode

The enhanced mode adds the following capabilities to the Auto-select tool:

- Modified scoring metrics
- Query operation
- Masking capability
- Overlap constraints

These capabilities are described in the following sections.

Modified Scoring Metrics

When run in non-enhanced mode, the Auto-select tool scores candidate window positions based on three metrics: symmetry, orthogonality, and uniqueness. In enhanced mode, the tool retains these metrics, however, there are some subtle differences in their evaluation.

A flag in the Auto-select parameters indicates whether the metrics should be evaluated using non-enhanced or enhanced algorithms. You can examine this flag during runtime, but you can set it at only the time the parameters object is constructed; you cannot change it later. The default is to use the non-enhanced algorithms.

Enhanced mode affects metric evaluations in the following ways:

- **Symmetry:** In enhanced mode, the symmetry score is based solely on the symmetry of the model window. It is not affected by the symmetry of nearby windows, as in non-enhanced mode. Therefore, the symmetry scores in enhanced and non-enhanced modes may differ significantly. In enhanced mode, symmetry scores are also evaluated on a higher resolution image than in non-enhanced mode. As a result, the sampling rate for optimal performance may be higher in enhanced mode than in non-enhanced mode. For example, if a sampling rate of 2 works best in non-enhanced mode, a sampling rate of 4 may work best in enhanced mode. The different sampling rates will cause the scores in the two modes to differ. However, most of the peaks in the symmetry score should occur at roughly the same window positions in both modes.
- **Orthogonality:** In non-enhanced mode, the orthogonality evaluation is tuned to the pixel grid. Scores decrease rapidly as the predominant edges in the image are rotated off of this grid. In enhanced mode, orthogonality scores should be similar to those of non-enhanced mode for axes-aligned images. However, in enhanced mode, the orthogonality score is insensitive to image rotation. High scores can still be achieved for window positions without axes-aligned edges, so long as there are a significant number of edges in each of two perpendicular directions.
- **Uniqueness:** Enhanced and non-enhanced mode use the same algorithm to evaluate uniqueness.

By default the Auto-select tool runs in non-enhanced mode, which is backward compatible with the original tool (provided with CVL 6.0 and earlier). You can run the tool in enhanced mode (in CVL 6.0.1 and later) by setting a flag when constructing the Auto-select parameters object. The tool may then compute different answers than in non-enhanced mode, but in general the answers should be similar. When the answers are significantly different, the answers produced in enhanced mode are probably preferable.

Note: The features described in the next sections (query operation, masking, and overlap constraints) are supported only when the Auto-select tool is run in enhanced mode. Attempts to use these features in non-enhanced mode will cause an exception to be thrown.

Query Operation

In non-enhanced mode, the Auto-select tool selects an appropriate position for a model window by considering all possible positions of the model window within the training image passed to the tool. This is the only mode of operation provided in non-enhanced mode. The enhanced mode retains this method, and refers to it as a *search operation*.

The enhanced mode has another search method, referred to as a *query operation*. For a query operation, you provide an explicit set of candidate window positions, and each of the positions is evaluated and scored using the same metrics as for a search operation.

Query operations are advantageous when you have some idea of the proper positions of the model windows, and need only select the best position among a small set of candidates. Running query operations is much faster than the exhaustive search performed for full search operations.

Query operations are also useful when the symmetry-centric approach used for search operations is not appropriate. With search operations, only window positions with relatively high symmetry are considered further, even if the weight attached to the symmetry score is small. If a particular window position has a low symmetry score, its orthogonality and uniqueness scores are never evaluated. This is done for efficiency. There is no way to make uniqueness, for example, the primary selection criterion. In a query operation, on the other hand, all three metrics are evaluated and reported at every specified window position. Hence, query operations allow you to implement more flexible selection algorithms.

Query operations are also useful for interactive applications. For example, they are appropriate for an application that allows a user to drag a model window around an image, reporting a quality measure for the given window position.

The programming interface for a query operation is similar to that for a search operation. The only difference is that you pass in one additional parameter: a vector of window positions at which to evaluate the scoring metrics. As with search operations, the tool returns a vector of result objects. The elements of this return vector are in one-to-one correspondence with the window positions passed in; they are not sorted according to total score, as with search operations.

For a query operation, you can disable the evaluation of particular scoring metrics by setting the corresponding weights to zero. If the weight is positive, the corresponding metric is computed.

When the Auto-select tool is run in enhanced mode, the computed scores of windows returned from a search operation are identical to those of the same windows when evaluated with a query operation. This is not generally the case when a search operation is performed in non-enhanced mode.

Masking

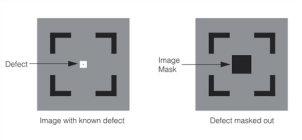
In enhanced mode, the Auto-select tool can ignore certain areas of an input image when calculating scores. Areas to be ignored are delineated with *masks*. A mask is an 8-bit pel buffer that masks out an area of an image or a window.

The Auto-select tool supports two basic types of masks: *image masks* and *window masks*. Image masks are further specialized into *position masks*. Both image and position masks are the size of the entire training image, while window masks are the size of the model window.

The next sections explain the different types of masks in detail.

Image Masks

Image masks are 8-bit pel buffers that are the size of the entire search image passed to the Auto-select tool. As they are the same size as the image, the location of image masks is fixed relative to the image search image. A possible application of image masks is to disregard known defects in an imperfect search image.



Example of image mask

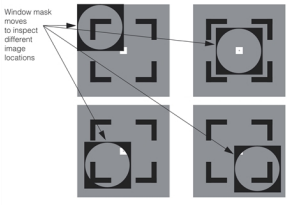
The image/position mask encodes three separate masks using three of eight available bits per pixel: image care mask, image don't score mask, and position mask. Together the first two comprise the image mask. See also *Mask Bit Codes* on page 368.

Position Masks

Position masks are also fixed relative to the entire search image passed to the Auto-select tool. During search operations, the Auto-select tool is prohibited from returning any window location that contains a pixel that has its position mask bit set. Hence, position masks may be used with search operations to create forbidden regions for good window positions. The position mask bit is ignored for query operations.

Window Masks

Window masks are 8-bit pel buffers that are the size of the model window. Window masks move with the search window as it scans over the entire search image. A possible application of window masks is to find an optimal circular region for model training, where the window size is based on the minimum enclosing rectangle of the circular region and the pixels outside the circle are masked.



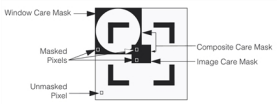
Example of window mask

Composite Masks

When evaluating a particular candidate window, the image and window masks are combined to form a *composite mask*. A pixel is considered masked if it is covered by any part of a composite mask in either the search image or the search window.

Composite Care Mask

The *image care mask* and *window care mask* are combined to form the *composite care mask*. Any pixel that is masked in either the image care mask OR the window care mask is considered to be masked in the composite care mask. For example, the highlighted pixels in the composite care mask in the figure below will not be included in the calculation of any score, whereas the unmasked pixel that is outside the composite care mask will be included. See also *Mask Bit Codes* on page 368.



Example of composite care mask

Composite Don't Score Mask

The *image don't score mask* and *window don't score mask* are combined to form the *composite don't score mask*. In contrast to the composite care mask, however, pixels must be masked in both the image don't score mask AND the window don't score mask to be considered masked in the composite don't score mask. The composite don't score mask is relevant only if PatMax is used as the pattern location tool; it affects only the uniqueness score, and is used only to evaluate clutter. See the chapter on *PatMax* on page 275 for more information on how clutter affects scores.

How Masking Affects Scores

Masking affects how individual scores are calculated.

Effect of Masking on Symmetry

The symmetry score is based on how well the image within the model window correlates to versions of the same image flipped horizontally and vertically. Masked pixels are ignored when this correlation is computed.

Only the composite care mask is used to calculate the symmetry score. The resulting correlation scores are combined and compared to the same scores at nearby window positions to arrive at a final symmetry score in the range of 0 to 1, inclusive. Pixels that do not have the composite care mask set are ignored in computing the raw correlation scores.

Effect of Masking on Orthogonality

The orthogonality score is based on the number of edgelets found within the search window that lie roughly along two perpendicular directions relative to the total number of pixels in the window.

Only the composite care mask is used to calculate the orthogonality score. Edgelets that lie within masked pixels for which the composite care mask is not set are excluded from the edgelet tally.

Effect of Masking on Uniqueness

The uniqueness score is computed by training the model from the training window, and then searching the entire input image for the trained model with the specified pattern- location tool. The score difference between the best match and the second best match determines the uniqueness score.

If you use PatMax as the pattern-location tool, the entire composite mask (that is, both the care and don't score layers) are passed to the PatMax training function. This is the only step of the entire Auto-select algorithm that uses the don't score mask. If you use CNLSearch as the pattern-location tool, the reverse of the composite care mask is passed to the CNLSearch training function. This is because CNLSearch uses the opposite convention from PatMax, using nonzero values to indicate masked pixels. It does not use the don't score mask. See the PatMax and CNLSearch chapters for details on how training time masking affects pattern location for these tools.

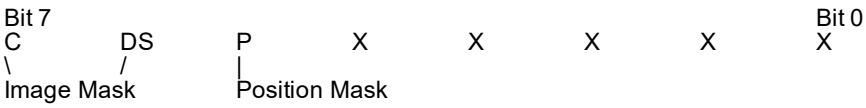
The Auto-select tool only supports training masks for PatMax; runtime masks are not passed to PatMax during uniqueness analysis.

Mask Bit Codes

The most significant bit (bit 7) of a pixel is the *care bit*, which determines whether the pixel is masked. Bit 6 is the don’t score bit, which determines whether the pixel should be scored as clutter. Bit 5 is the *position mask bit*, which determines whether the position belongs to a forbidden region for search operations. The following sections describe in detail the meanings of mask bits.

Image Masks

Bit assignments of image/position mask pixels are as follows:



C	=Care bit	1 = Care (unmasked)	0 = Don’t care (masked)
DS	=Don’t Score bit	1 = Don’t score	0 = Score
P	=Position mask bit	1 = Position masked	0 = Not position masked
X	=Ignored		

The DS bit is used only with PatMax during uniqueness score evaluation, and it is significant only if the C bit is cleared. The combination 10 in the upper two bits is illegal in a PatMax image and will cause an exception to be thrown. When using PatMax for pattern location, if the C bit is set then make sure the DS bit is also set. When using CNLSearch for pattern location, the DS bit is ignored regardless of the state of the C bit.

Window Masks

Bit assignments of window mask pixels are as follows:

C	DS	X	X	X	X	X	X

C =Care bit — 1 = Care (unmasked) — 0 = Don't care (masked)
DS =Don't Score bit — 1 = Don't score — 0 = Score
X =Ignored

Notice that there is no position mask bit for window masks.

The same information regarding the C and DS bits as described for image/position masks also applies to window masks.

Overlap Constraints

In enhanced mode, Auto-Select tool allows you to limit the amount of overlap between model windows returned from a search operation. The overlap between two same-sized windows is approximately the ratio of the area of their intersection to the area of one of the windows. The overlap constraint is enforced for every pair of windows returned from a search operation. Setting a maximum overlap of 0.0 forces the tool to return completely disjoint windows. Setting a maximum overlap of 1.0 effectively disables overlap checking. The overlap constraint does not apply for query operations.

Using the Auto-Select Tool

This section explains how to use the Auto-select tool.

General Guidelines

Observe the following general guidelines when using the Auto-select tool:

- Use an input image that is similar to the actual runtime images your application will encounter.
- Specify a square, or nearly square, model window

Note: The Auto-select tool will not return model locations that lie within a 32-pixel-wide boundary around the edge of the input image.

Specifying the Mode

The Auto-select tool runs in non-enhanced mode by default (see *Modes of Operation* on page 363). To run the Auto-select tool in enhanced mode, set the *enhancedMode* parameter of the **ccAutoSelectParams** constructor to true when you create the parameters object. For example:

```
ccAutoSelectParams params(true);
```

If you do not provide an argument to the **ccAutoSelectParams** constructor, or provide a value of false, the Auto-select tool will run in non-enhanced mode.

You cannot change the operating mode once the parameters object has been created. If you want to change the mode, you must construct a new parameters object.

To determine whether the tool is running in enhanced mode during runtime, query the parameters object with **ccAutoSelectParams::isEnhancedMode()**. For example:

```
cogOut << "Auto-select tool is running in " ;
 if(params.isEnhancedMode())
 cogOut << "enhanced mode" << endl;
 else
 cogOut << "non-enhanced mode" << endl;
```

Specifying the Model Size

Specify a model size that is related to the size of the features of interest within the image. If the model size is too small, the Auto-select tool will be unable to locate a region that contains enough of the feature of interest. If the model size is too large, performance will be reduced.

Note: Depending on the pattern-location tool you are using, there may be a minimum or maximum model size for the tool.

Specifying the Sub-Sampling Factor

In general, avoid specifying a sub-sampling factor greater than 8. As a starting point, examine your input image and note the approximate size of the smallest meaningful features within the image. For a given sub-sampling factor, features that are smaller in size than the sub-sampling factor tend to be lost.

Specifying PatMax and CNLSearch Parameters

When you specify PatMax or CNLSearch runtime parameters, make sure that they correspond to the parameters that you will be using to search for the pattern or model returned by the Auto-select tool. Observe the following specific guidelines:

- For CNLSearch, specify the same CNLSearch algorithm that you will be using to search for models. The Auto-select tool will return very different candidate models for different algorithms.
- For PatMax, specify the same nominal and enabled degrees of freedom and zones that you will be using in your application. Patterns can vary widely in their uniqueness depending on which degrees of freedom are enabled.

Specifying the Score Computation Method

Observe these guidelines when specifying the score computation method:

- Specify the *geometric mean* if you want a high overall score to reflect high scores for each of the individual score components.
- Specify the *arithmetic mean* if you want a high overall score to reflect high scores for all component scores *or* a high score for a single component score.

Specifying Masks

You pass an image or position mask to the Auto-select tool as a parameter to the **cfAutoSelect()** function. However, you pass a window mask within a **ccAutoSelectRunParams** object.

For information on the distinctions between various types of masks, see *Image Masks* on page 366 and *Position Masks* on page 367, and see *Window Masks* on page 367.

Specifying the Overlap Value

The degree to which candidate windows can overlap is relevant only for search operations. If you specify an overlap value for query operations, it will be ignored.

Interpreting Scores

The Auto-select tool *always* attempts to return a suggested region for model or pattern training. If the image that you supply to the Auto-select tool does not contain any regions with high orthogonality and high symmetry, the Auto-select tool may return candidates with good uniqueness scores that are, in fact, poor candidates. In general, if a model or pattern candidate has low symmetry and orthogonality scores but a high uniqueness score, it may not be an ideal candidate.

Wafer Pre-Align Tool

The Wafer Pre-Align tool is a vision tool that determines the coarse location and orientation of silicon wafers. The Wafer Pre-Align tool can locate wafers based on a model that you supply, or it can automatically learn appropriate wafer parameters based on an input image.

This chapter includes the following sections:

Some Useful Definitions on page 372 defines some terms that you will encounter as you read.

What the Wafer Pre-Align Tool Does on page 372 describes the Wafer Pre-Align tool's major features and applications.

How the Wafer Pre-Align Tool Works on page 372 describes how the Wafer Pre-Align tool works.

Using the Wafer Pre-Align Tool on page 376 provides some guidelines for using the Wafer Pre-Align tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

alignment feature: A feature of a silicon wafer that can be used to determine the orientation of the wafer. Wafer flats and wafer notches are the two alignment features recognized by the Wafer Pre-Align tool.

coarse alignment: Determining the approximate location and orientation of an object.

What the Wafer Pre-Align Tool Does

The Wafer Pre-Align tool determines the location and orientation of a silicon wafer. This information is used typically used to re-position the wafer before reading a string, bar code, or 2D symbol from the wafer.

Wafer Types

The Wafer Pre-Align tool can locate three types of wafers: round wafers, wafers with an alignment flat (flat wafers), and wafers with an alignment notch (notched wafers). The figure below shows the appearance of each type of wafer.

Wafer types

The Wafer Pre-Align tool can determine the location of all of these wafer types; the tool can determine the orientation only of flat and notched wafers.

Wafer Alignment Overview

You can use the Wafer Pre-Align tool in the two ways. You can specify the expected diameter and wafer type along with the expected dimensions of the alignment feature, then have the tool determine the orientation and location of wafers in run-time images. You can also have the Wafer-Pre Align tool automatically learn the wafer size and wafer type based on an image that you supply.

How the Wafer Pre-Align Tool Works

The way that the Wafer Pre-Align tool locates wafers depends on how you configure and run the tool.

Operating Modes

The Wafer Pre-Align tool supports two operating modes, *NotchMax mode* and *Standard mode*. Each of these modes is described in this section.

NotchMax Mode

NotchMax mode is a robust, highly accurate operating mode that allows you to accurately align wafers using front lighting. NotchMax mode is simple to use; very few run-time parameters are required. NotchMax mode requires that the entire perimeter of the wafer be visible without extraneous features.

In NotchMax mode, the tool finds wafers by locating the wafer boundary in the image and matching it to the wafer dimensions you provide.

Standard Mode

Standard mode is intended for applications where a back-lit silhouette image of the wafer can be acquired. In addition to requiring back-lit images, Standard mode requires that you provide detailed, accurate information about the expected size of the alignment notch or flat and about the expected eccentricity of the wafer image. Standard mode tolerates some extraneous features or missing sections along the wafer perimeter.

In Standard mode, the tool finds wafers by searching for patterns in the input image that match a synthetic model of the wafer that you supply.

Performance Considerations

For notched wafers, NotchMax is faster than Standard mode. For flat wafers, Standard mode is faster (and may be more accurate at locating the wafer center).

Licensing

Each Wafer Pre-Align tool operating mode has an associated security bit. In order to use an operating mode, the corresponding security bit must be enabled in the Cognex hardware device you are using. When you construct a Wafer Pre-Align tool object (type **ccWaferPreAlign**) , you specify the operating mode for that object. The table below shows the effect of specifying different operating modes with different license bits set.

Requested Mode	Security Bits		Actual Mode
	NotchMax	Standard	
NotchMax	Yes	Yes	**NotchMax**
	Yes	No	**NotchMax**
	No	Yes	Exception thrown
	No	No	Exception thrown
Standard	Yes	Yes	**Standard**
	Yes	No	Exception thrown
	No	Yes	**Standard**
	No	No	Exception thrown
Auto-Detect	Yes	Yes	**NotchMax**
	Yes	No	**NotchMax**
	No	Yes	**Standard**
	No	No	Exception thrown

Wafer Pre-Align tool operating modes and security bits

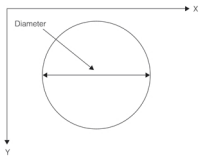
Training a Model Wafer

The first step in using the Wafer Pre-Align tool in Standard mode is to train a synthetic model of the wafer you want to locate. You supply the tool with the expected size of the wafer and the wafer type. This section describes each of the parameters you can set.

Note: The parameters described in this section do not apply to NotchMax mode.

Wafer Diameter

You specify the expected diameter of the wafer in client coordinate system units as shown in the figure below.



Specifying expected wafer diameter

Image-to-Client Transformation

The training-time transformation that you supply to the Wafer Pre-Align tool describes how you expect actual images of wafers to vary from the ideal model wafer. In almost all cases, you should specify the transformation object that you obtain when you calibrate your camera and optical system. This transformation, in addition to defining the real world units of measurement (usually millimeters), will indicate the aspect ratio of your camera.

Learning Wafer Parameters

The Wafer Pre-Align tool can automatically determine appropriate values for the following parameters:

- Wafer diameter
- Wafer type (round, notched, or flat)

To have the tool automatically learn these training parameters, you supply an image of a typical wafer to your application and invoke the learn function. You can specify the wafer type and diameter, if they are known, or you can have the tool determine them automatically.

As part of the learning process, the Wafer Pre-Align tool will automatically locate and measure the wafer in the learning image that you supply, using the run-time parameters that you supply.

Note: If you are using Standard mode and you request that the Wafer Pre-Align tool automatically learn the feature type, you should make sure to specify a range of expected feature sizes for both flat features and notch features.

Locating The Trained Wafer

Once you have either manually trained or automatically learned appropriate training parameters, you invoke the Wafer Pre-Align tool by supplying it with an input image of a wafer and a set of run-time parameters. Some run-time parameters are used in both Standard and NotchMax operating modes while others are only used in Standard mode.

Common Run-Time Parameters

This section describes the run-time parameters used in all operating modes.

Wafer Type

You must specify the wafer type (round, notched, or flat) when you search for a wafer. The Wafer Pre-Align tool can automatically learn the wafer type based on an input image.

Expected Scale Range

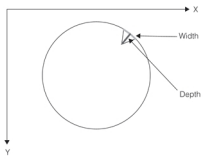
At run-time you specify the expected scale range for the wafer in the run-time image. The Wafer Pre-Align tool will only find a wafer if its scale varies from the trained wafer by no more than the range of scale values that you specify.

Standard Mode Run-Time Parameters

This section describes the run-time parameters that are used only in Standard mode.

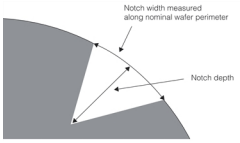
Expected Notch Dimensions

For wafers with an alignment notch, you specify the expected notch width and depth. *Specifying expected wafer diameter on page 374* shows the parameters that you specify for a notched wafer.



Specifying expected notch dimensions

Note: The width of a notch is measured along the nominal perimeter of the wafer; the width of a notch is *not* the minimum distance between the two corners of the notch. The notch depth is measured from the deepest point of the notch to the nominal wafer perimeter along the wafer radius. The figure below shows how the notch width and depth are measured.

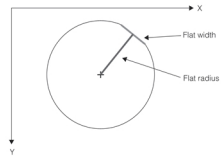


Notch width and depth

Note that you can specify the expected minimum and maximum depth and width for the notch. Wafers with notches whose dimensions are outside of the ranges you specify will not be found by the tool.

Expected Flat Dimensions

For a wafer with an alignment flat, you specify the expected flat width and radius. The figure below shows the parameters you specify for a flat wafer.

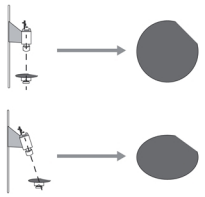


Specifying wafer flat dimensions

Note that you can specify the expected minimum and maximum flat width and flat radius for the flat. Wafers with flats whose dimensions are outside of the ranges you specify will not be found by the tool.

Eccentricity

For all wafer types you can specify the expected degree of eccentricity of the wafer. While all wafers are round, if your camera is not perpendicular to the wafer, then foreshortening will cause the wafer to appear elliptical, as shown in the figure below. Ordinarily, this would cause problems in attempting to align the wafer.



Foreshortening causes elliptical-appearing wafers

You can specify the maximum expected degree of eccentricity for the Wafer Pre-Align tool. The eccentricity *e* is computed using the following formula: $e = \sqrt{1 - \frac{min^2}{max^2}}$

where *min* is the length of the minor principal axis and *max* is the length of the major principal axis. A value of 0.0 indicates a perfect circle.

Accept Threshold

All wafer instances located by the tool are assigned a score value to indicate how closely the wafer in the image resembles the model wafer. The score can range from 0.0 (a poor or nonexistent match) to 1.0 (a perfect match).

When you invoke the tool, you must specify an *accept threshold*. This value is the minimum score that you expect an actual wafer instance to receive. In most cases, the default value for this threshold works well.

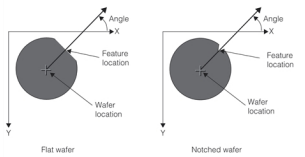
Wafer Pre-Align Tool Results

For each image that you supply to the Wafer Pre-Align tool, the tool attempts to locate a single instance of the model wafer within the image. For each image, the tool returns the following results:

- Whether the wafer was found
- Whether the specified alignment feature was found
- The type of alignment feature that was found
- The size of the wafer, expressed as the scale between the model wafer and the found wafer
- The dimensions of the alignment feature, if one was found
- The location and orientation of the wafer, and the location of the alignment feature (if one was found)

The feature dimensions are described in the section *Locating The Trained Wafer* on page 374.

The wafer location is considered to be the location of the wafer center. The feature location is the deepest point of the notch (for notched wafers) or the center of the flat (for flat wafers). The wafer's orientation is the angle from the client-coordinate system x-axis to a line drawn from the wafer center through the feature location. *Foreshortening causes elliptical-appearing wafers* on page 375 shows how these measures are returned.



Wafer angle and location and feature location

Note: For round wafers, the tool returns no angle, no feature location, and no feature dimensions.

Using the Wafer Pre-Align Tool

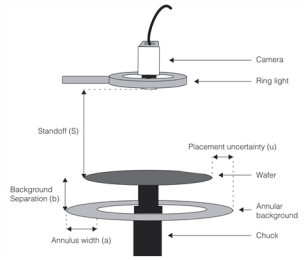
This section provides some guidelines for using the Wafer Pre-Align tool. For more information, refer to the programming reference documentation for your framework.

Input Image Requirements

The input image requirements are different for the two Wafer Pre-Align tool operating modes. This section describes the input image requirements for both modes.

NotchMax Mode Image Requirements

NotchMax mode is intended to support images acquired using front lighting where the wafer is placed in front of an annular background. The figure below shows a typical configuration.



NotchMax configuration

The dimensions of this configuration should be within the limits listed in the table below.

Dimension	Minimum Value	Maximum Value
Wafer Size	150 mm 200 pixels	300 mm 400 pixels
Placement uncertainty (u)		5 mm
Standoff (S)	300 mm	3 m
Background separation (b)	1 mm	1 m
Annulus width (w)	$\left(2u+3\right)\frac{s+b}{s}$	

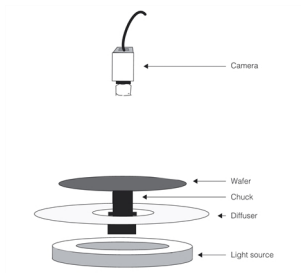
NotchMax configuration dimensional limits

In addition to the dimensional requirements listed in the table above, images acquired for use in NotchMax mode should conform to the following guidelines:

- There must be no occlusion of the wafer boundary; the entire perimeter of the wafer must be visible.
- The Resolution of the acquired image must be approximately 0.75 mm per pixel.
- The annular background should be of uniform intensity.

Standard Mode Image Requirements

Standard mode is intended to support images acquired using back lighting. The figure below shows the typical configuration for use in Standard mode.



Standard configuration

Any input image you use for learning or for wafer alignment in Standard mode should meet the following requirements:

- The wafer should be backlit and should appear as a silhouette
- The wafer should occupy about 90% of the area of the image

Manual Training

If you are specifying the model wafer parameters manually, you should make sure that you specify a range of dimensions for the wafer size and feature size that will encompass all of the wafers you expect to encounter during your application.

Automatic Learning

In general, if you know the expected wafer diameter and wafer type, you should specify these parameters when you learn a wafer.

Image Registration Tool

This chapter describes the Image Registration tool. It contains the following sections:

This section and *Some Useful Definitions on page 379* provide an overview of the chapter and define some terms that you will encounter as you read.

Image Registration Tool Overview on page 379 provides information about the capabilities and intended use of the Image Registration tool.

How the Image Registration Tool Works on page 380 provides a general description of the operation of the Image Registration tool.

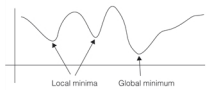
Using the Image Registration Tool on page 382 describes some of the techniques that you will use to implement an application using the Image Registration tool.

Some Useful Definitions

image registration: A search technique designed to determine the exact point at which two images of the same scene are precisely aligned.

global minimum: The lowest value in a function or signal.

local minima: A low value in a function or signal.



subpixel accuracy: Positions within an image may be specified in terms of whole pixel positions, in which case the position refers to the upper-left corner of the pixel, or in terms of fractional pixel positions, in which case the position may lie anywhere within a pixel. Positions specified in terms of fractional pixel positions are referred to as having subpixel accuracy.

Image Registration Tool Overview

The Image Registration tool performs fast, accurate image registration using two images that you supply.

Image Registration

Image Registration is the process of determining the precise offset between two images of the same scene.

You use the Image Registration tool to perform image registration by providing a reference image and a source image, along with a starting point within the source image. The reference image should be a reference image of the feature you are attempting to align. The source image should be larger than the reference image, and it should contain an instance of the reference image within it.

The Image Registration tool determines the precise location of the reference image within the source image. The Image Registration tool will return the location of this match, called the *registration point*, with subpixel accuracy.

The figure below illustrates an example of how the Image Registration tool performs image registration.

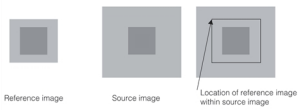


Image Registration

Note: The registration point is defined to be the location of the origin of the reference image within the source image's *image coordinate system*.

The Image Registration tool is optimized to locate precisely the reference image within the source image, even if the source image contains image defects such as reflections, or if the pattern from the reference image is partially obscured by other features in the source image.

The figure below illustrates an example of image registration where the source image is partially obscured.

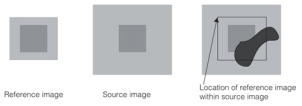


Image Registration with partially obscured source image

How the Image Registration Tool Works

The Image Registration tool finds the location of the reference image within the source image. You operate the Image Registration tool by supplying the reference and source images, along with the location within the source image where you expect the origin of the reference image to be. The Image Registration tool will determine, with subpixel accuracy, where the origin of the reference image lies within the source image.

The Image Registration tool works by computing a score indicating the degree of similarity between the reference image and a particular portion of the source image that is the same size as the reference image. The tool computes this score for locations in the immediate neighborhood surrounding the starting point. The tool will find the location within this neighborhood of the source image that produces the local peak in the value of this score.

By adding an interpolation step, the Image Registration tool determines the location of the reference image within the source image with subpixel accuracy. For typical images, the Image Registration tool can achieve accurate registration to within 0.25 pixels.

Because of the way the Image Registration tool seeks the local score peak, if the starting point you specify is more than a few pixels from the actual registration point, the tool may not return the correct registration point. The exact amount of variance that the Image Registration tool can tolerate will vary depending on the images. The variance may be as small as 3 to 5 pixels for some images or as large as 30 pixels for others.

Image Registration Scoring

In addition to returning the location of the reference image within the source image, the Image Registration tool also returns a score indicating the degree to which that portion of the source image matches the reference image.

The Image Registration tool provides two scoring methods: a *sum of absolute differences* method and a *normalized correlation* method. Each of these methods is described in this section.

Note: Regardless of which scoring method you use, if your image registration receives a score indicating a poor match, the actual precision of the image registration location may be somewhat lower than if the image registration receives a score indicating a closer match.

Sum of Absolute Differences Scoring

The sum of absolute differences method computes a score by summing the absolute differences of all of the pixels in the reference image with the corresponding pixels in the source image. Using this method, a score of 0 indicates a perfect match. Nonzero scores indicate a mismatch, with larger scores indicating poorer matches.

Normalized Correlation Scoring

The normalized correlation method computes the normalized correlation coefficient between the pixel values in the reference image and the values of the corresponding pixels in the source image. The formula for normalized correlation coefficient r of the reference image and the corresponding portion of the source image at image offset (u,v) is given by

$$r\left(u,v\right)=\frac{\left[N\sum_i I_iM_i-\left(\sum_i I_i\right)\left(\sum_i M_i\right)\right]}{\sqrt{\left[N\sum_i I_i^2-\left(\sum_i I_i\right)^2\right]\left[N\sum_i M_i^2-\left(\sum_i M_i\right)^2\right]}}$$

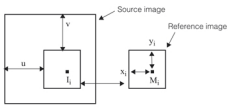
where

N is the total number of pixels.

I_i is the value of the source image pixel at $(u+x_i, v+y_i)$.

M_i is the value of the corresponding reference image pixel at the offset (x_i, y_i).

The figure below shows the relationship among these components.



A reference image and the corresponding portion of the source image

Using the normalized correlation method, a score of 1.0 indicates a perfect match between the images. A score of 0.0 indicates an unmatched image while negative scores indicate a mismatched image.

Sub-Pixel Accuracy

The Image Registration tool determines the sub-pixel location at which the normalized correlation coefficient is the highest between the reference image pixels and the run-time image pixels. You can select the method that the tool uses to compute this sub-pixel location:

- *Standard* mode uses a basic mathematical model for estimating the sub-pixel position with the highest score.
- *High-Accuracy* mode computes the sub-pixel location of the correlation score peak using a mathematical model that better characterizes how the normalized correlation coefficient changes at small sub-pixel image offsets. (Standard mode uses a mathematical model that may not accurately model changes in the correlation score at small image offsets.)

Exhaustive Image Registration

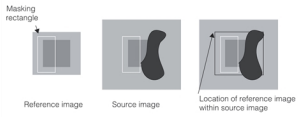
When you use the Image Registration tool, you supply the tool with two images and a starting location within the reference image. The tool will confine its image registration search to a small area around the starting location that you specify.

The Image Registration tool can also perform image registration exhaustively, that is, by computing the score for every possible location of the reference image within the source image. This procedure, called *exhaustive image registration*, is extremely slow. It can be helpful, however, in debugging applications where the Image Registration tool does not appear to be working correctly.

Masked Image Registration

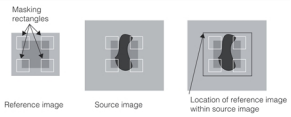
You can limit the areas of the reference image and source image that the Image Registration tool uses to perform the image registration by supplying one or more masking rectangles to the tool. If you supply these rectangles, the tool will compute the score based only on those pixels contained within the rectangles that you specify.

The figure below illustrates an example of specifying a masking rectangle. In this example, the right side of the pattern in the reference image is often obscured in the source image. By specifying a rectangle that covers the left side of the reference image, you can cause the Image Registration tool to consider only the pixels in that part of the reference image. This will tend to increase the accuracy of the image registration.



Masked image registration

You can also specify several masking rectangles. The figure below illustrates an example where the center of the reference image is often obscured in the source image. By specifying a series of rectangles, you can eliminate from consideration the part of the image that is most frequently obscured.



Masked image registration using multiple rectangles

If you specify multiple masking rectangles and the rectangles overlap, any pixels that are contained in more than one masking rectangle will be counted toward the score multiple times, once for each rectangle in which they are contained.

In all cases, the Image Registration tool will return the location of the origin of the reference image within the source image.

Using the Image Registration Tool

This section describes how to use the Image Registration tool.

Obtaining Reference and Source Images

When you acquire the reference image, you should take care to ensure that the image is as ideal a model as possible of the images that you will be using as source images. The Image Registration tool is designed to align source images with a wide variety of image defects, but in order for the tool to work, the *reference* image needs to be as free from defects as possible.

Specifying a Starting Point

When using the Image Registration tool, you must specify a starting location in the reference image. The tool will perform its image registration search starting at the point that you specify. Because the tool only seeks the local minima for the score value, if you specify a starting location that varies greatly from the actual registration point, the image registration operation will fail.

You can use other pattern location tools such as PatMax or CNLSearch to locate the pattern from the reference image within the source image, or you can rely on operator input to specify the coarse location of the feature.

PatInspect

This chapter describes PatInspect, Cognex software that uses PatMax technology to detect and report defects from the acquired image of an inspection object.

This chapter contains the following sections:

The first two sections, this one and *Some Useful Definitions on page 383*, provide an overview of the chapter and define some terms that you will encounter as you read.

PatInspect Overview on page 383 provides an introduction to PatInspect.

Training on page 385 describes how PatInspect computes template regions and other statistics from a given sample of training images. Note that although PatMax can be trained using a synthetic model or an image model, PatInspect supports only image models.

Run-time Inspection describes how PatInspect finds and reports differences between template regions and inspection regions.

Using PatInspect on page 404 describes the techniques that you use to implement an application using PatInspect.

Note: You should already be familiar with PatMax before reading this chapter or using PatInspect.

Some Useful Definitions

alignment region: A region used by PatInspect to align training or run-time images with a reference image.

mean region: A region whose pixels are the mean of the pixel values computed across analogous regions in the sample of training images.

inspection region: A region to be inspected in the training or run-time images. Inspection regions can be specified as either pel buffers or affine rectangles.

reference image: An image in which the alignment and inspection regions are defined. Training and run-time images are aligned with this image.

region: A rectangular portion of an image.

run-time region: A region, within the run-time image, to be inspected.

standard deviation region: A region whose pixels are the standard deviation of the pixel values computed across analogous regions in the sample of training images.

statistical training: Training from a sample of acceptable instances of the image that is going to be inspected at run-time.

template region: Synonym for mean region.

threshold region: A region derived from a standard deviation region, used to find defects in the run-time image.

PatInspect Overview

The purpose of PatInspect is to detect defects. PatInspect creates the template image of an object from a sample of training images and then compares run-time images of the object against the template image. For PatInspect, a defect is any change in the run-time image that is beyond the expected variation in the training sample. A defect can be an erroneous or unwanted mark on an object, an incorrectly shaped feature, or the absence of a feature.

PatInspect detects defects according to three distinct modes:

- Intensity difference inspection
- Feature difference inspection
- Blank region inspection

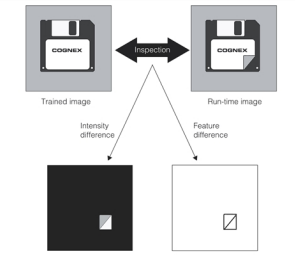
Intensity Difference Inspection

Intensity difference detects differences in grey-scale pixel value between the trained image and a run-time image. Intensity difference includes support for image normalization so that grey-scale pixel value differences caused by lighting variations are not marked as differences.

Feature Difference Inspection

Feature difference detects defects based on difference in feature position, shape, or size between the trained image and the run-time image. Feature difference mode is immune from both uniform and non-uniform intensity differences. Feature difference is also capable of detecting sub-pixel-sized defects.

The figure below shows an example of the intensity differences and feature differences between two images. The folded corner of the diskette label creates two regions within the image that have different intensity levels. These regions are apparent in the intensity difference image. The folded label corner also creates a new feature not present in the trained image (the folded corner) and removes a feature that was present in the trained image (the unfolded corner). These two features appear as feature differences.



Intensity and feature difference

You can use PatInspect to detect intensity differences, feature differences, or both kinds of differences.

Blank Region Inspection

Blank region inspection detects defects in image areas that are supposed to be blank. Patinspect defines blankness as the absence of trainable boundary features.

Operational Overview

The operation of the PatInspect vision tool consists of two distinct phases:

- Training
- Run-time inspection

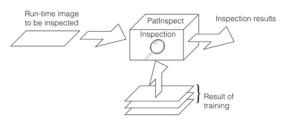
Overview of Training

During this phase you supply PatInspect with a sample of acceptable instances of the image that you want to inspect at run-time. For intensity difference and blank scene inspection the output of this phase consists of a template image, a standard deviation image and a threshold image. For feature difference inspection the output is just a template image (see the figure below).

Overview of training

Overview of Run-time Inspection

During run-time inspection PatInspect uses the information acquired during training to inspect a run-time image (see the figure below).



Overview of run-time inspection

Each of these two phases is described in more detail in the next sections.

Training

The training phase consists of two major steps:

- Region selection.
- Statistical training (for intensity difference and blank scene mode only).

Note: Although PatMax can be trained using a synthetic model or an image model, PatInspect supports only image models.

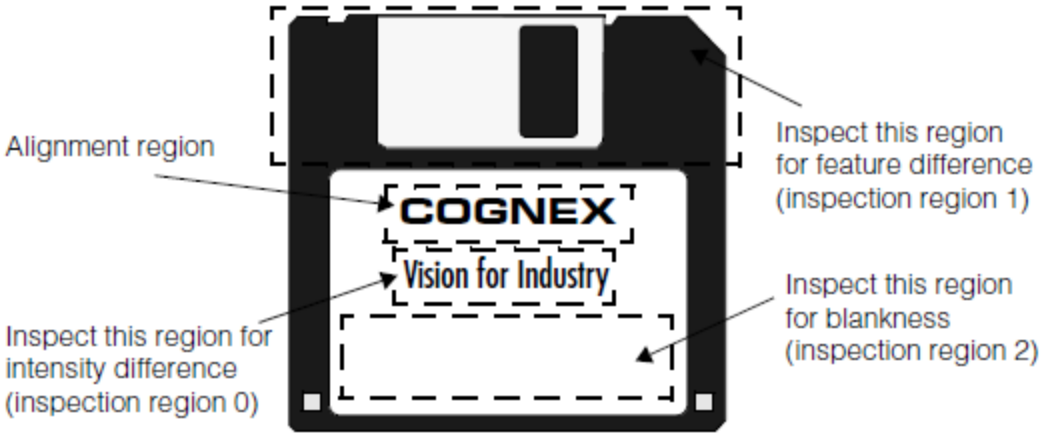
Region Selection

In this step you tell PatInspect what regions within the training images must be trained and in what mode. You start from a *reference image* such as the one shown in the top figure below and you partition the image into several *inspection regions* as shown in the second figure below. You can use both pel buffers and affine rectangles as inspection regions. When you partition the reference image you must:

- decide which region will be used for alignment
- assign an inspection mode to each region (intensity difference, feature difference or blank region inspection)

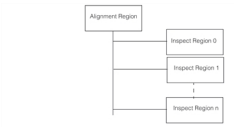
In the second figure below the region containing the string “COGNEX” is the alignment region, one region is selected for blankness inspection, one region is selected for intensity difference inspection and one for feature difference inspection.

The reference image



Regions to be inspected within the reference image

All inspection regions are internally organized by PatInspect as a list (see the figure below).



Internal organization of the alignment and inspection regions

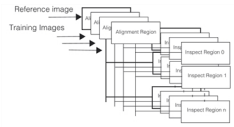
At the top of the list is *always* the alignment region. This is the region used by PatInspect to align all training and inspection images with the reference image. When you present PatInspect with a training or inspection image, the tool performs the following set of operations to align them with the reference image:

1. PatInspect locates the alignment region within the training or inspection image by using PatMax technology.
2. PatInspect gets the 2D transformation that maps the alignment region in the training or run-time image to the alignment region in the reference image. This transformation is called the *pose*.
3. PatInspect uses the pose to align the training or run-time image to the reference image.

There are instances in which you may already know the alignment pose for your application. In this case, you can directly supply PatInspect with the pose and skip the step of finding the alignment region.

Statistical Training

In this step you present PatInspect with a sample of acceptable instances of the images you want to inspect and let the tool compute some statistics from the sample. Each time you present a training image, PatInspect will align it with the reference image and add it to the stack of all the other training images as illustrated in the figure below.



The training images are aligned and internally organized as a list of stacked regions

At the end of the training process, PatInspect goes trough the stack of training images and for each region computes:

- the mean region
- the standard deviation region
- the threshold region

The mean region is the region representative of all the sample regions. This region is called the *template region*. The set of all template regions forms the *template image*.

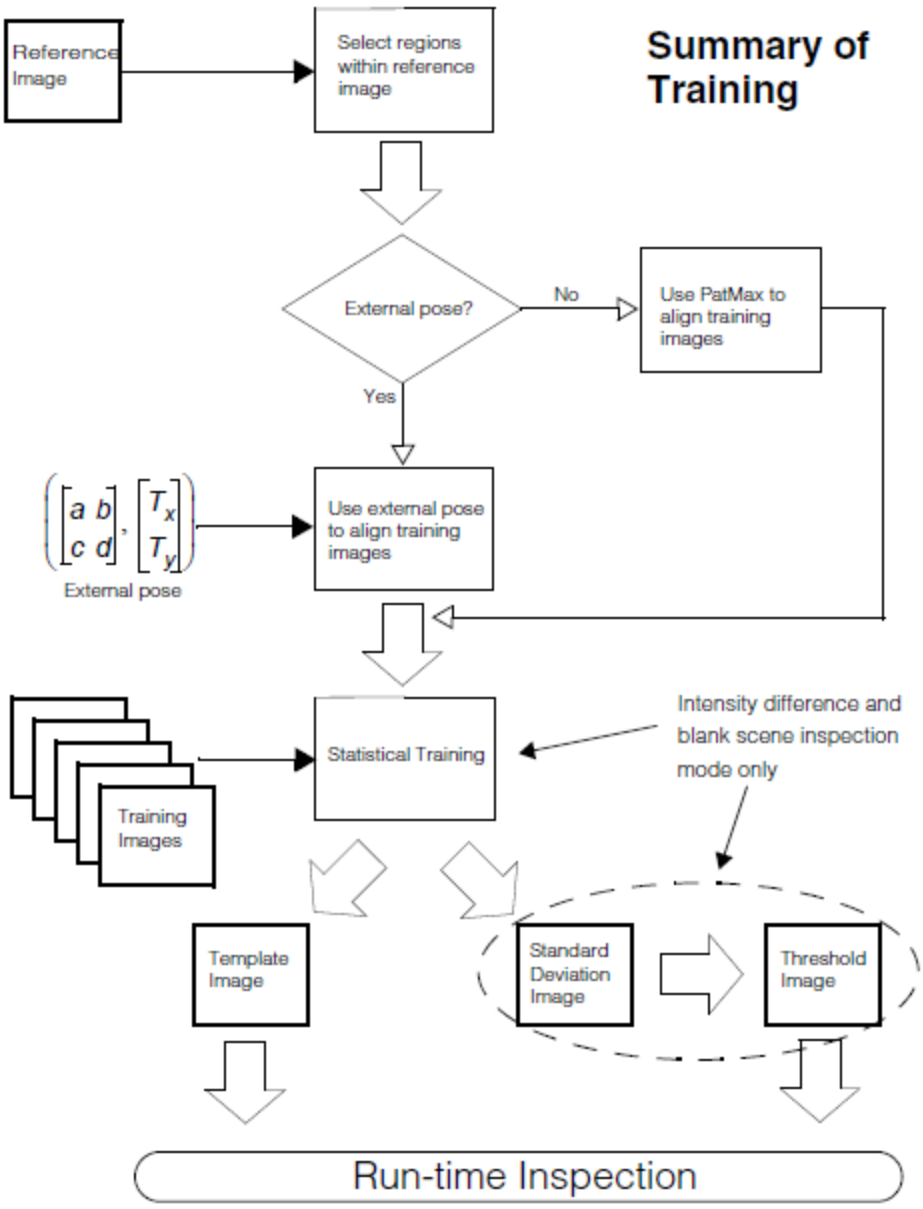
The standard deviation region is the region that accounts for the degree of variability within the sample. The set of all standard deviation regions forms the *standard deviation image*.

The threshold region is derived from the standard deviation region and is used during inspection to determine the presence of defects. The set of all threshold regions forms the *threshold image*.

Statistical training is currently available only for intensity difference mode and blank scene inspection mode. Training for boundary difference does not take into account the statistical properties of the sample of training images (see *Training for Feature Difference* on page 392).

Note: PatInspect allows you to enable or disable training regions. The number of iterations used to compute the statistics from the sample of training images can then be different from region to region.

The figure below summarizes PatInspect training. The next sections provide a more detailed description of training for each of the three inspection modes.



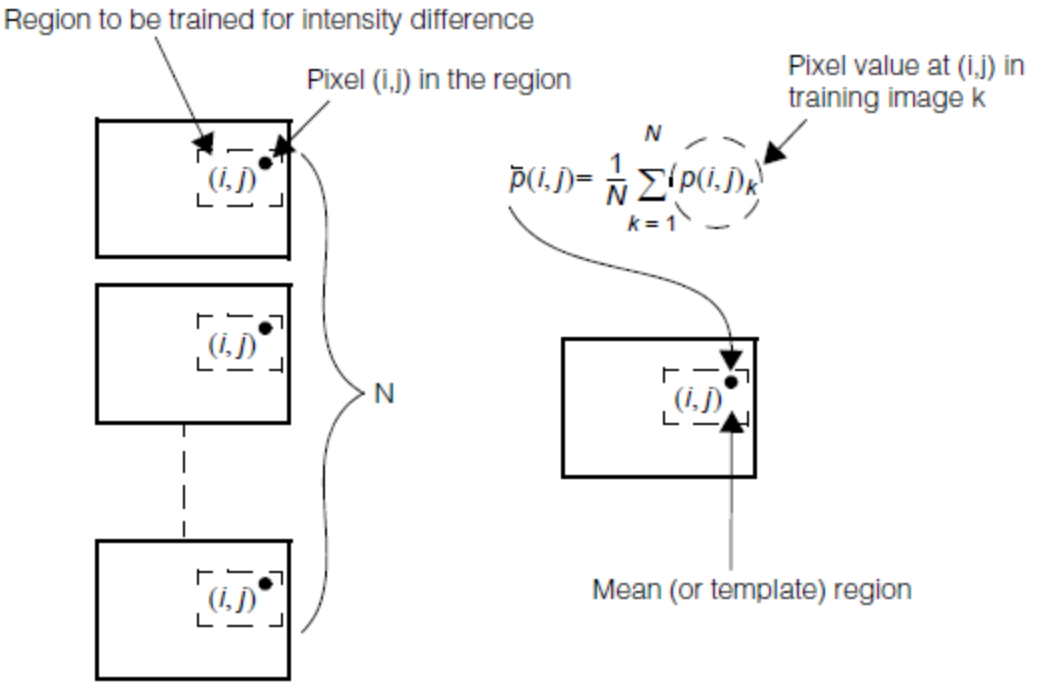
Summary of training

Training for Intensity Difference

This section describes how the mean, standard deviation and threshold regions are computed for intensity difference training.

Mean Region

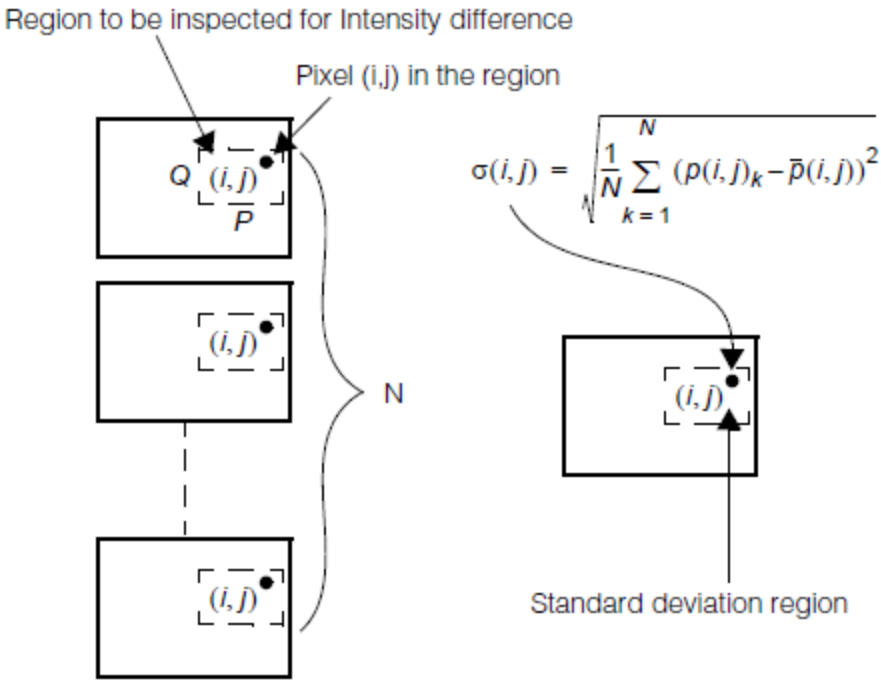
If you are training for intensity difference, each pixel value in the mean region is equal to the mean pixel value computed across the N training regions as illustrated in the figure below.



The mean (or template) region for intensity difference training

Standard Deviation Region

Each pixel value in the standard deviation region is equal to the standard deviation of the pixel values computed across the N training regions as illustrated in the figure below.



The standard deviation region for intensity difference training

A high value of $\sigma(i,j)$ means high variability of the pixel (i,j) in the sample of training images.

Threshold Region

The threshold region is derived from the standard deviation region by setting each pixel value to the following:

$$(Threshold)_{(i,j)} = A \times \sigma(i,j) + B$$

Where (i,j) are the pixel coordinates, $\sigma(i,j)$ is the standard deviation of the pixel at (i,j) while A and B are constants that you can set. This image is used during run-time inspection to evaluate the likelihood that a pixel is a defect (see *Thresholding the Difference Region* on page 395).

PatInspect lets you also create a *synthetic* threshold image from a single training image. The synthetic threshold image is obtained from the convolution of the training image with a Sobel edge-detection filter. Pixel intensity values in the Sobel-filtered image are higher in the presence of edges. Since regions with edges are more likely to vary from image to image, the Sobel-filtered image can also be used as an approximation to a statistically derived threshold image.

Training for Blank Scene Inspection

This section describes how the mean, standard deviation and threshold regions are computed for blank scene training.

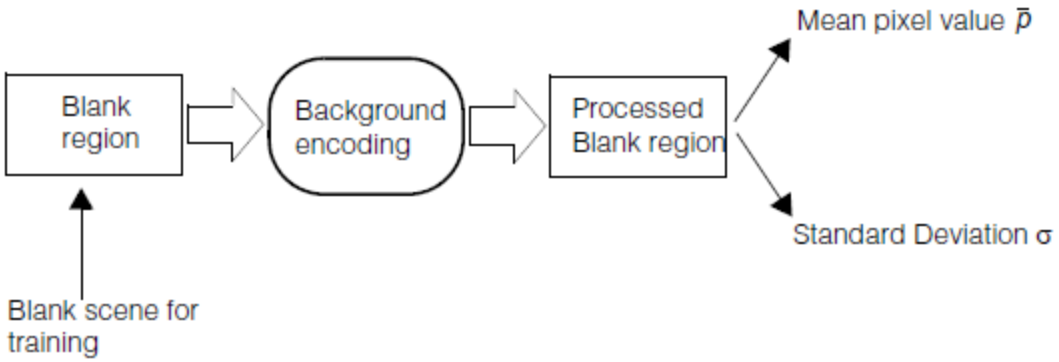
Mean Region

The computation of the mean region for blank scene inspection involves several steps:

1. The training blank scene region is first processed to encode the background of the region (see the figure below).
2. The mean pixel value and the standard deviation of the processed blank scene are derived. If the size of the blank scene is (PxQ) the two statistics are:

$$\bar{p} = \frac{1}{P \times Q} \sum_{i=1}^{P} \sum_{j=1}^{Q} p(i,j) \quad \text{and} \quad \sigma = \sqrt{\frac{1}{P \times Q} \sum_{i=1}^{P} \sum_{j=1}^{Q} (p(i,j) - \bar{p})^2}$$

where $p(i,j)$ is the pixel value of the processed image at the pixel location (i,j) (see the figure below).



The pre-processing of a training blank scene

The values of $\bar{p}$ and σ are computed for each blank scene in the set of training images. If you have supplied N training images, PatInspect computes the following parameters:

$\overline{P} = \frac{1}{N} \sum_{i=1}^{N} \bar{p}_i$ and $\overline{\sigma} = \frac{1}{N} \sum_{i=1}^{N} \sigma_i$, where $\bar{p}_i$ and σ_i are the mean pixel value and standard

deviation of the blank scene region in the training image *i*. If the training region has size (PxQ), the mean region is the region of size (PxQ) in which all pixels are set to $\overline{P}$ (see the figure *Summary of blank scene training* on page 392).

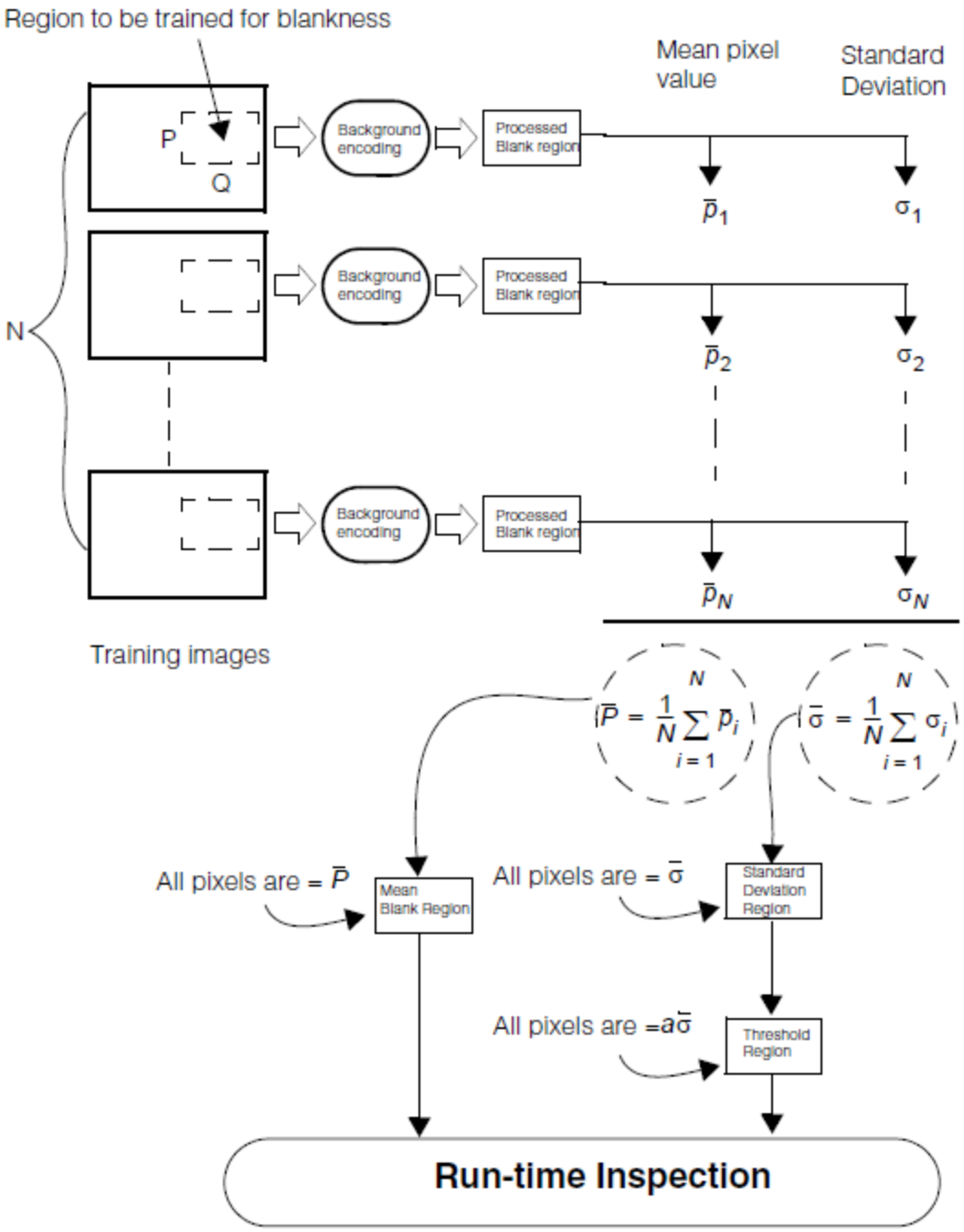
Standard Deviation Region

The standard deviation region is the region of size (PxQ) in which all pixels are set to $\overline{\sigma}$ (see the figure *Summary of blank scene training* on page 392). PatInspect allows you to override the value of $\overline{\sigma}$ produced during training.

Threshold Region

The threshold region is the region of size (PxQ) in which all pixels are set to $a\overline{\sigma}$, where a is a parameter that you can adjust (see the figure below).

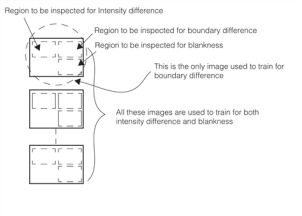
All the steps involved in blank scene training are summarized in the figure below.



Summary of blank scene training

Training for Feature Difference

Training a region for feature difference inspection does not currently involve the computation of any statistics from the sample of training images. The template region, against which the run-time region is compared, is formed by the boundary features obtained from the first training region supplied to PatInspect for training. All the other regions are currently neglected as illustrated in the figure below. This limitation will be removed in a future release.



Only one image is used for boundary difference training

Untraining

When you untrain PatInspect, the internal histogram used to compute the values for histogram equalization is reset to zero. If your application untrains and then retrains using histogram equalization, the equalization values will be different.

Run-Time Inspection

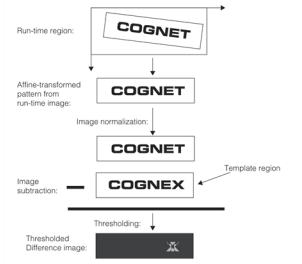
This section describes how PatInspect finds and reports defects in the run-time inspection image for all three inspection modes.

Intensity Difference Inspection

PatInspect intensity difference inspection follows these steps:

1. PatMax locates the alignment region in the run-time image unless you provide an external pose.
2. The run-time region is affine transformed to be aligned with the corresponding template region.
3. The affined transformed run-time region is normalized to the template region using a method that you specify.
4. The correlation coefficient between the template region and the normalized run-time region is computed.
5. The normalized run-time region is subtracted from the template region to produce a difference region
6. The threshold region is applied to the difference region

The figure below provides a summary of the steps involved in PatInspect intensity difference inspection. In the example the template region contains the string “COGNEX” while the corresponding run-time inspection region contains the string “COGNET”.



PatInspect intensity difference inspection

Localization of the alignment region

PatInspect uses the standard PatMax tool to locate the alignment region in the run-time image. You can bypass the PatMax pattern location technology by supplying your own pose to PatInspect. For each pose that you supply, PatInspect allows you to compute the difference image and the correlation coefficient.

Affine Transformation

You can specify the type of sampling (nearest neighbor, bilinear interpolation, or high-precision) to be used by PatInspect to affine-transform the run-time inspection region. For more information on these sampling types, refer to the chapter *Image Transformation Tools* on page 109. The affine transformed run-time region is aligned with the template region.

Difference Region

Once PatInspect has aligned the run-time inspection region with the corresponding template region, it performs the following steps to compute the difference region:

1. It normalizes the run-time region to the pixel values in the template region.
2. It creates a region that is the absolute value of the difference between each pixel in the template region and the normalized run-time region.
3. It thresholds the difference region.

These steps are described in more detail in the next sections.

Normalizing Run-Time Images

PatInspect performs grey-scale normalization on the run-time inspection region to compensate for normal changes in lighting and aperture between training and inspection. Normalization is performed by *pixel mapping*. The pixel map is derived at run time using statistics calculated during training: histogram, mean, standard deviation, and left and right tails. The pixel map is calculated by a method that you can select. Four calculation methods are available; they are listed below in order according to the size of defects typically expected in your application.

The PatInspect tool lets you select from the following normalization methods:

- *Histogram equalization*: Derive the pixel map from the histogram calculated during training.

  A histogram of the run-time region is generated and compared to the histogram calculated during training. The mapping is constructed so that the histograms match. (That is, at any bin in the histogram of the run-time region, the total number of pixels with a value less than or equal to that bin number equals the total number of pixels in the training histogram with such a value, within a round-off tolerance).

  Histogram equalization is a powerful, general-purpose form of grey-scale normalization. It is appropriate where the total defect area is fairly small, or where the typical defect amplitude is small. It is well suited where lighting or optics variations can lead to nonlinear grey-scale variations between training time and run time. However, histogram equalization may be inadequate when the defect area is large enough to alter substantially the histogram distribution.

- *Mean and standard deviation*: Derive the pixel map from the mean and standard deviation of the pixel grey-scale values calculated during training.

  The mapping is constructed so that the mean and standard deviation of the normalized run-time region will match the mean and standard deviation from training. This effectively equalizes the contrast of the images.

  Mean and standard deviation is appropriate for images with moderately sized defects.

- *Tails matching*: Derive the pixel map from the left and right tails calculated during training.

  The *tails* are outlying points (representing grey-scale values) on the left and right ends of a histogram beyond which grey-scale values are considered unreliable; grey-scale values outside the tails are discarded before the mapping is done. The range of grey-scale values between the tails of the test image's histogram is linearly mapped to the range between the tails of the training histogram.

  Tails matching is especially well suited for situations in which there are likely to be large defects that substantially alter the shape of the histogram but not its range.

- *Identity transformation*: Map each grey-scale value to itself; no change in the image.

Computing the Difference Region

After normalization, PatInspect computes the difference region defined as the absolute value of the difference between the pixels in the run-time region and the corresponding pixels in the template region. The resulting difference region contains pixels with values between 0 (no difference in value between the two regions) and the maximum pixel value allowed (255 for 8 bit images).

Thresholding the Difference Region

The difference region inevitably contains non-zero values in most of its pixels. When training in intensity difference mode, PatInspect automatically constructs a threshold region as described in *Training for Intensity Difference* on page 388. This region has high pixel values for pixels in the training regions that exhibited a lot of variability and low pixel values for pixels that didn't vary much. When PatInspect applies the threshold region to the difference region, only pixels with values that exceed the threshold are passed through. The pixels that survive the thresholding indicate differences that are too high to be simply attributed to random noise variations in the run-time image. Such differences are, therefore, very likely to be due to the presence of defects in the run-time image of the object to be inspected.

Intensity Difference Results

PatInspect compute the following result information:

- The thresholded difference region
- The correlation coefficient between the template region and the run-time region.

The correlation coefficient (r) is computed using the following formula:

$$r = \frac{\left[N \sum_i \sum_j I(i,j) M(i,j) - \left(\sum_i \sum_j I(i,j)\right)\left(\sum_i \sum_j M(i,j)\right)\right]}{\sqrt{\left[N \sum_i \sum_j I(i,j)^2 - \left(\sum_i \sum_j I(i,j)\right)^2\right]\left[N \sum_i \sum_j M(i,j)^2 - \left(\sum_i \sum_j M(i,j)\right)^2\right]}}$$

where

N is the total number of pixels in the region

$I(i,j)$ is the value of the pixel at (i, j) in the affine transformed run-time region.

$M(i, j)$ is the value of the corresponding pixel at (i, j) in the trained region.

Other Returned Information

In addition to the thresholded difference image, PatInspect returns all the location and scoring information returned by PatMax. You can obtain the same precise information about how the run-time alignment region varies from the alignment region in the template image as you can with PatMax.

Note: If you supply your own pose, then the returned pose will be the same as the pose you supply and the score values will be 1 and the clutter score, contrast score, fit error, and coverage score will be -1. PatInspect returns one set of results for each alignment region found in the run-time image.

Using Thresholded Difference Images

For most applications, you will use grey-scale morphology and blob analysis to process the thresholded difference region. You can use the information returned by the Blob tool to extract and classify defects found in the run-time region.

Feature Difference Inspection

This section describes how PatInspect performs inspection using feature differences. PatInspect follows these steps:

1. PatMax locates the alignment region in the run-time image and aligns it with the reference image (unless external pose is provided).

2. PatInspect generates detailed information about how the feature boundaries vary between trained and run-time inspection regions.

Computing Feature Difference Information

PatInspect computes feature difference information by comparing boundary features within the run-time region and the template region. PatInspect lets you control how features are detected and matched in the run-time region by letting you specify the following parameters:

- A match quality low threshold and a match quality high threshold

  You can specify that only boundary points with a match quality above a threshold be considered. Match quality is a measure of both the strength of the feature boundary point and its proximity to a feature in the trained image.

  Features with match quality above the high threshold are considered. Features with match quality between the low and high thresholds are discarded. Features with match quality below the low threshold are treated as either missing or extraneous.

- A minimum feature size

  You can specify that only features larger than a specified size be considered.

- A minimum feature contrast value

  You can specify the minimum contrast in grey levels for feature boundary points in the image. Only those feature boundary points in the image with contrast values greater than the threshold you specify are considered by the tool.

- A boundary deformation parameter

  You can specify the distance in pixels within which feature boundary points can be matched. This parameter indicates how far away a feature boundary point in the run-time region can be from a feature boundary point in the template region and still be considered for a match.

- An ignore polarity parameter

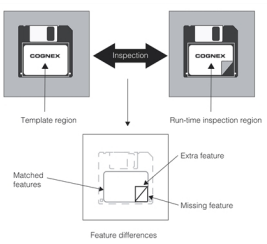
  You can specify whether to ignore the polarity of the run-time region during inspection.

Returned Feature Difference Information

When you request feature difference information from PatInspect, PatInspect computes and returns three lists of features:

- The *match list*, which contains all of the features that are present in the template region and in the run-time region
- The *missing list*, which contains all of the features that are present in the template region but that are missing from the run-time region
- The *extra list*, which contains all of the features that are present in the run-time region but that are not present in the template region

The figure below shows examples of features that would appear in each of these lists.



Matched, missing, and extra features

Feature Information

PatInspect returns the following information about each matched, missing, and extra feature:

- A list of the feature boundary points that make up the feature
- Whether the feature is open or closed
- The maximum and minimum deformation of a boundary point within the feature (matched features only)
- The average weight and match quality of the boundary points in the feature

For each boundary point in the list that makes up a feature, PatInspect returns the following information:

- The location of the boundary point in the run-time image.
- The location of the corresponding boundary point in the training-time image (matched features only)
- The weight of the point, which is a measure of the strength of the feature boundary at that point
- The match quality of the point, which is a measure of how well the point in the run-time image matches the corresponding point in the training image

Additional Feature Information

PatInspect lets you compute additional information about any individual feature from any of the returned feature lists. You can compute the following additional information about any feature:

- The feature's segments
- The feature's mousebites
- The feature's mean position
- The feature's length
- The feature's area
- The feature's minimum enclosing polygon
- The feature's bounding rectangle

Each of these items is discussed in the following sections.

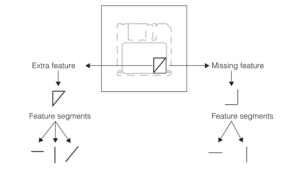
Feature Segments

When PatInspect computes a feature, it does so by assembling all the neighboring feature boundary points that make up the feature into a list. Features can be of any size and shape, and they can be open or closed.

You can segment PatInspect features based on angle change between neighboring feature boundary points and minimum segment size (number of boundary points). If you specify a minimum feature segment size, feature segments with fewer than the specified number of boundary points are discarded.

Segmenting a feature based on angle change lets you break a feature into feature segments at points of high curvature. For example, if you specify an angle of 45°, then wherever the feature boundary has an angle of greater than 45° between adjacent feature boundary points, the feature is broken into segments.

The figure below shows the effect of segmenting the missing and extra features detected in the figure *Matched, missing, and extra features* on page 396 in the section Returned Feature Difference Information using an angle of 45°.



Segmenting features

You can specify a minimum feature segment size when you segment a feature.

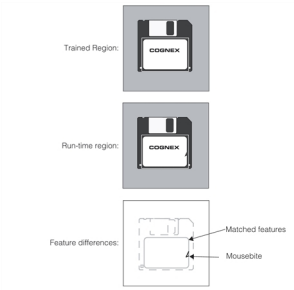
You can obtain the size, average weight, and average match quality for each feature segment in a feature.

Mousebites

You can use PatInspect to detect relatively small, localized feature differences called *mousebites* within features. A mousebite is defined as a set of boundary points within a matched feature that have the following attributes:

- The boundary points are contiguous within the feature
- The distance between the boundary points in the run-time image and the corresponding points in the trained image is at least: $k \times Std$
  where k is a parameter that you can set and Std is the standard deviation of the distances of all boundary points from their corresponding positions in the trained image.
- The number of boundary points is greater than a minimum size that you specify.

The figure below shows a trained image, a run-time image, and the feature boundary points within a feature that PatInspect would identify as a mousebite.



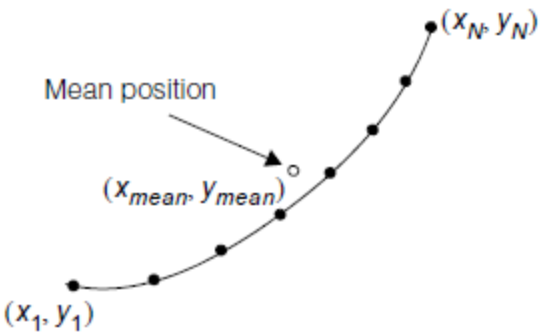
Mousebite

By computing mousebites you can distinguish between discrete defects and normal pattern variations.

Feature's mean position

The mean position of a feature is the position whose x- and y- coordinates are the mean of the x- and y- coordinates of all the boundary points that make up the feature. If the
feature is made up by N points, the x-and y- mean positions of the feature are:

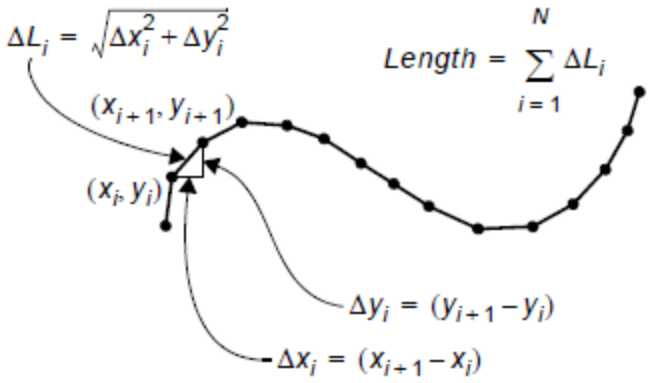
$x_{mean} = \frac{1}{N}\sum_{i=1}^{N} x_i$ and $y_{mean} = \frac{1}{N}\sum_{i=1}^{N} y_i$ (see the figure below).



The mean position of a feature

Feature's Length

PatInspect returns the length of the feature boundary in the client coordinates that you have specified. The figure below illustrates how the length of the boundary feature is computed (assume the feature has N points).

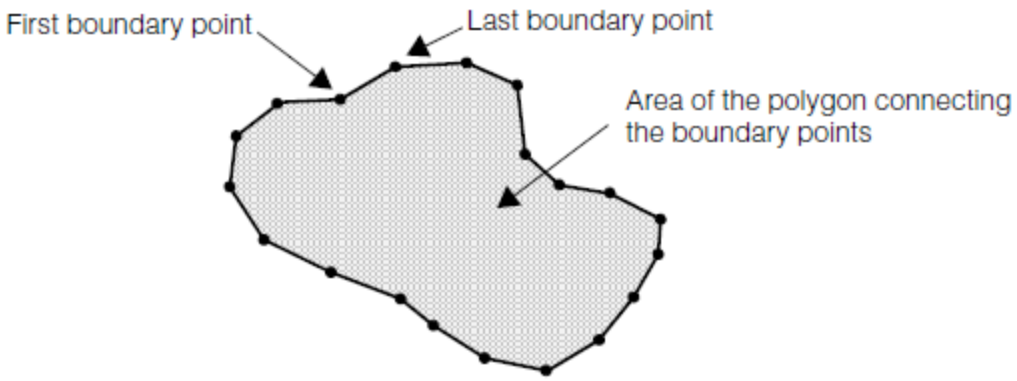


How the length of a feature boundary is computed

Feature's area

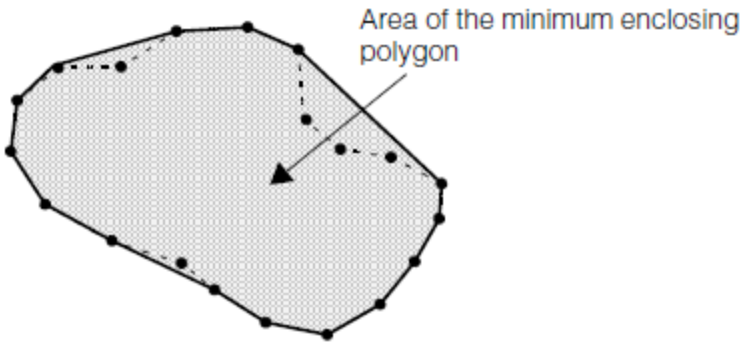
PatInspect allows you to compute the area enclosed by a feature. If the feature is open the area returned is the one of the closed feature boundary obtained by joining the first and last point of the boundary with a straight segment. PatInspect can perform two different area measurements:

- It measures the area of the closed polygon obtained by joining the boundary points by line segments as illustrated in the figure below.



The area of the polygon that connects the feature's boundary points

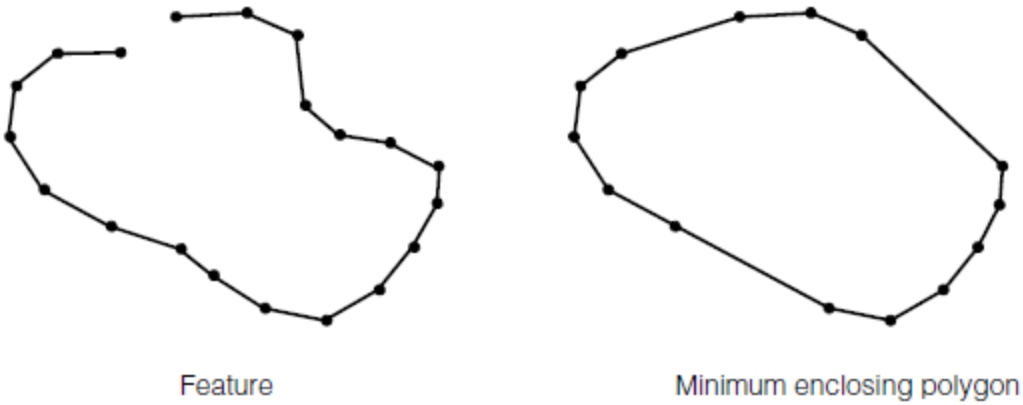
- It measures the area of the minimum enclosing polygon as illustrated in the figure below.



The area of the boundary feature's minimum enclosing polygon

Feature's Minimum Enclosing Polygon

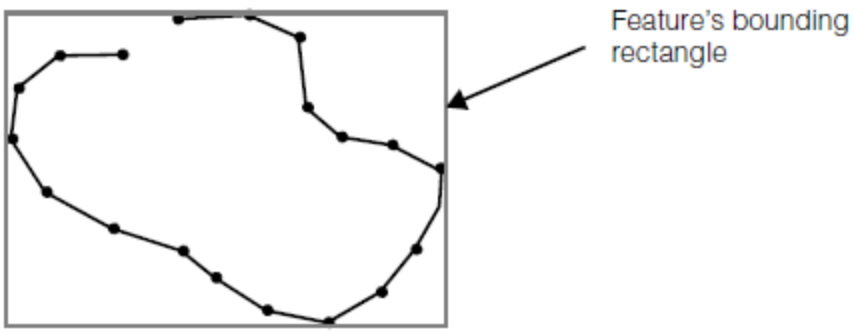
PatInspect returns the minimum enclosing polygon of a feature as illustrated in the figure below.



The minimum enclosing polygon of a feature

Feature's Bounding Rectangle

PatInspect returns the feature's bounding rectangle as illustrated in the figure below.



The feature's bounding rectangle

Other Returned Information

In addition to feature difference information, PatInspect returns all of the location and scoring information returned by PatMax. You can obtain the same precise information about how the run-time alignment region varies from the alignment

region in the template image as you can with PatMax.

Note: If you supply your own pose, then the returned pose will be the same as the pose you supply and the score values will be 1 and the clutter score, contrast score, fit error, and coverage score will be -1. PatInspect returns one set of results for each alignment region found in the run-time image.

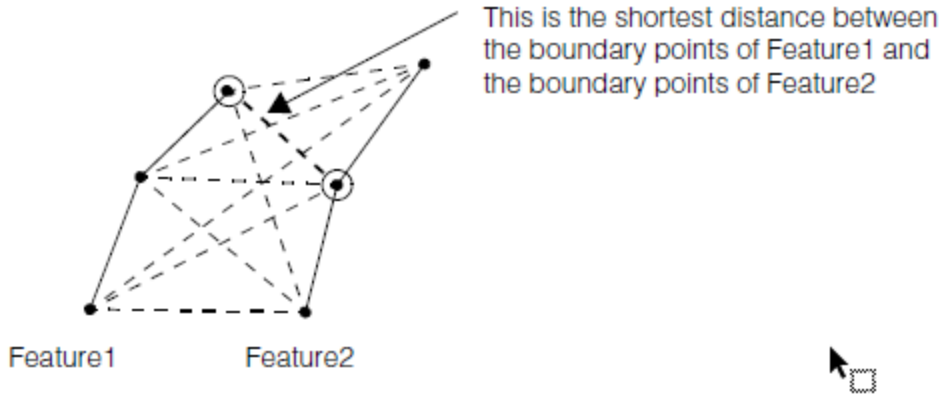
Feature Boundary Post-Processing

Once the match, missing and extra feature boundaries have been extracted from the run-time inspection region, you can ask PatInspect to determine the distance and angle between two of them.

Distance Between Two Features

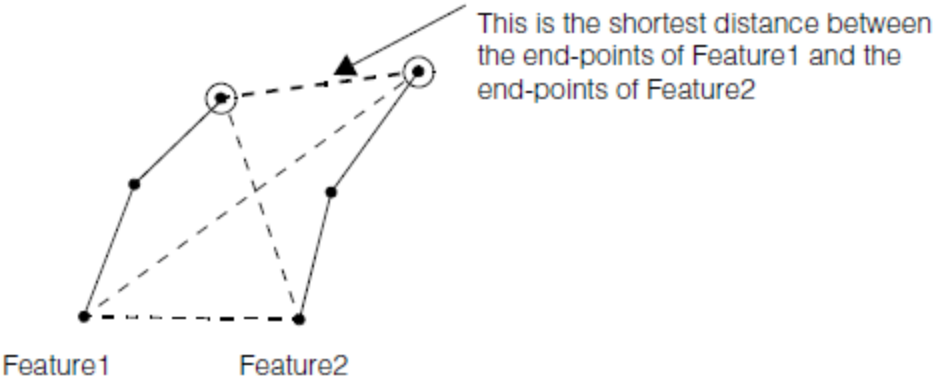
PatInspect can measure three types of distances between two features:

- The shortest distance between the boundary points of two features as illustrated in the figure below.



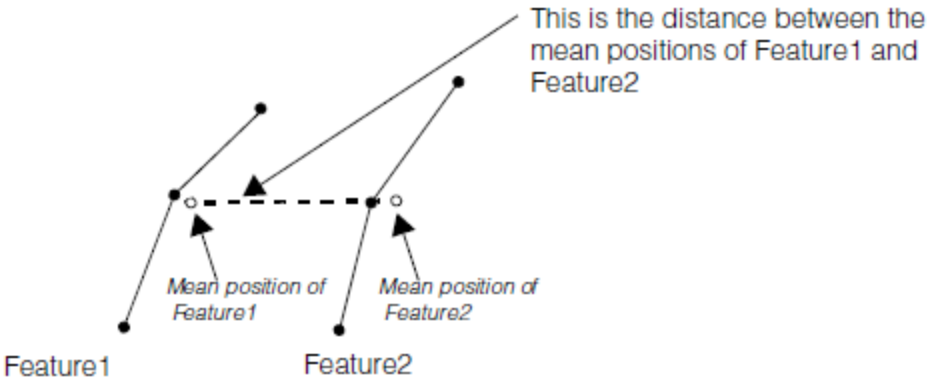
The shortest distance between two features

- The shortest distance between the end-points of two features as shown in the figure below.



The shortest distance between the end-points of two features

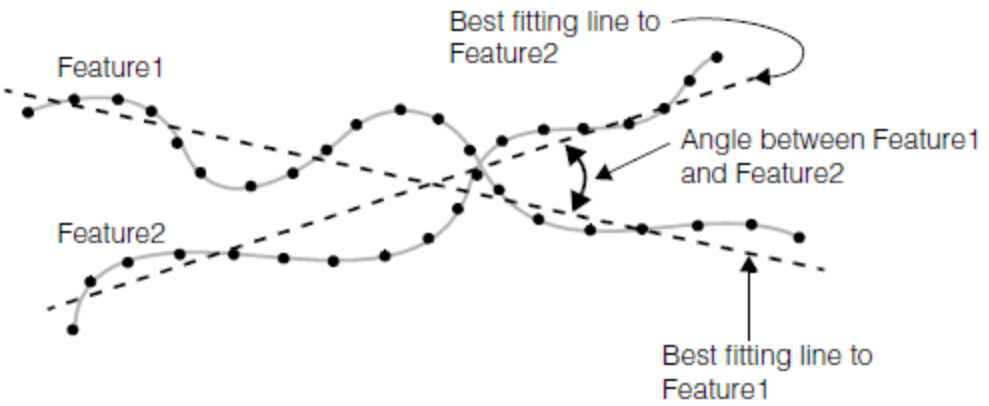
- The distance between the mean positions of Feature1 and Feature2 as illustrated in the figure below.



The distance between the mean positions of two features

Angle Between Two Features

PatInspect allows you to compute the angle between two features. The angles between two feature boundaries is defined as the angle between the best fitting lines to the feature boundaries as illustrated in the figure below.



The angle between two feature boundaries

Software Zoom for Boundary Feature Inspection

PatInspect includes a boundary feature inspection modality that allows you to train and inspect regions that contain very thin or densely packed features. The problem with these regions is that, after feature detection, the boundaries from the thin and densely packed features may blur together and turn out to be inaccurately resolved.

Software zoom magnification is a technique that allows under-resolved boundary features to be spatially separated in a higher resolution image. The basic idea underlying the technique is to consider the pixels of the under-resolved image as samples from an hypothetical higher resolution image and reconstruct the higher resolution image by pixel interpolation. The effect of software zoom magnification is to spatially separate the unresolved boundaries of image features to permit accurate feature detection.

You can use software zoom magnification in two ways:

- You can use software zoom for pattern training but not at run-time inspection. In this case boundaries in the training region are detected at a higher resolution. The resulting boundaries are then scaled back to the original resolution to form the template region. The run-time inspection region is not software zoomed and is compared with the template region.

- You can use software zoom both at training and run-time. In this case both the run-time and template regions are compared after software zooming. The match, missing and extra boundary features are obtained after software zooming and then scaled back to the original resolution.

Note: You may need to adjust the training-time contrast threshold slightly as you increase the training-time software zoom. Make sure you use the diagnostic display to evaluate the feature selection during training.

The first software zoom mode increases the training time but has no run-time effect. The second software zoom mode is more accurate but takes slightly longer to run because of additional run-time processing.

Note: You specify a software zoom value by providing a fine granularity value of less than 1.0. Keep in mind that granularity (described in the section *Feature Size and Pattern Granularity* on page 279) is the value that PatMax and PatInspect use to detect features. By specifying a granularity value of less than 1 pixel, you are indicating the radius of interest is less than 1 pixel.

Fine Alignment for Boundary Feature Inspection

PatInspect allows you to refine the alignment of the regions that are to be inspected in feature difference mode. This is a second fine alignment after the first main alignment performed by locating the alignment region in the inspection image. This second fine alignment affects only the inspection region and is performed by aligning the fine boundary features of the region. This property of PatInspect is useful in all those situations where slight spatial distortions in the inspection region may have occurred. You can retrieve the refined pose if the fine align mode is enabled.

Blank Scene Inspection

This section describes how PatInspect performs blank scene inspection. PatInspect follows the following steps:

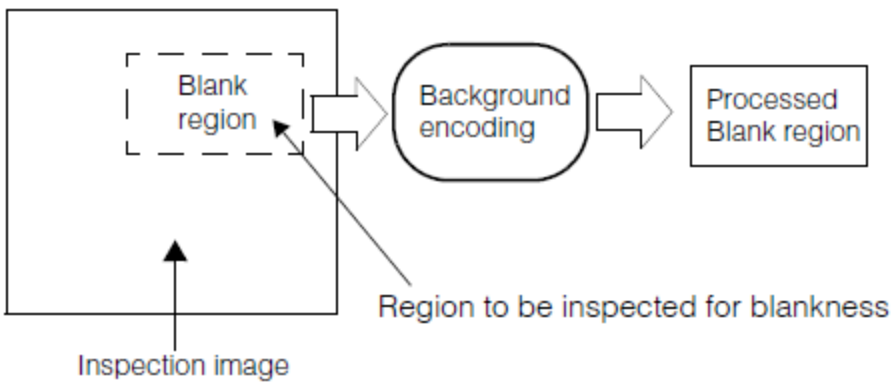
1. It identifies potential defect areas
2. It localizes defects within these areas

Each of these steps is discussed in more detail in the next sections.

Identifying Potential Defect Areas

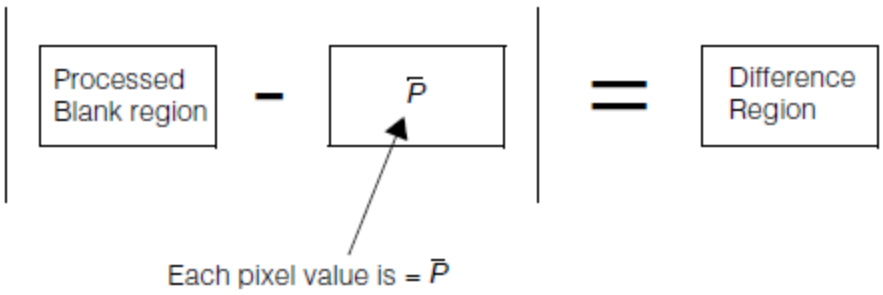
The purpose of this step is to identify those areas within the inspection region that are likely to contain defects:

1. The region is processed to encode the background (see the figure below)



The region to be inspected for blankness is processed to encode the background

2. The difference region is obtained. The difference region is the absolute value of the pixel by pixel difference between the processed region and the template region (see the figure below and *Training for Blank Scene Inspection* on page 390).

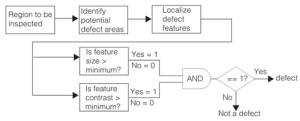


Difference region for blank scene inspection

3. Each pixel in the difference region is compared with the threshold region (see*Training for Blank Scene Inspection* on page 390). The pixels in the difference region that are greater than threshold are set to 255 and the pixels that are less than threshold are set to zero. The non-zero areas within the region are where boundary extraction and defect localization is performed.

Localization of Defects

The boundary features within the non-zero areas of the thresholded difference region are considered potential defects. If the size and contrast of these features exceed the minimum feature size and minimum feature contrast values that you have set, they are labeled as defects. The figure below summarizes all the steps involved in the identification of defects in a blank region.



Summary of blank scene inspection

Returned Blank Scene Information

PatInspect returns the list of the extra boundary features that have been labeled as defects.

Using PatInspect

This section describes how to use PatInspect.

PatInspect Training Guidelines

This section provides some guidelines you should follow when training PatInspect.

Alignment Training Region

When you choose the alignment region you should avoid including features in the training image that may not be present in the search image.

Pattern Granularity Limits

Like PatMax, PatInspect automatically determines appropriate coarse and fine granularity limits. See the section *Feature Size and Pattern Granularity* on page 279 for more information on pattern granularity.

Intensity Difference Training Regions

If possible, you should supply multiple intensity difference training images. Supplying images that exhibit a normal range of pattern variation helps PatInspect construct an effective threshold region and can reduce spurious differences.

PatInspect Parameter and Result Summary

Training

The table below summarizes the training parameters and results for each inspection mode.

Type	Inspection mode	Item	Notes
Training parameters	Intensity difference	Two coefficients to compute threshold region	Used to obtain the threshold region from the standard deviation region (see *Threshold Region* on page 390)
		Sobel coefficients	The coefficients of the Sobel filter used to obtain the synthetic standard deviation region (see *Threshold Region* on page 390)
	Boundary difference	None	None
	Blank scene inspection	Standard deviation value	The standard deviation value for blank scene inspection (you can override the standard deviation computed from the sample, see *Threshold Region* on page 391)
		One coefficient to compute threshold region	Used to obtain the threshold region from the standard deviation region (see *Threshold Region* on page 391)
Training results	Intensity difference and Blank scene inspection	Template image Standard deviation image Threshold image	The threshold image is obtained from the standard deviation image. The template image and the threshold image are used during run-time inspection
	Boundary difference	Template image	The standard deviation and threshold image are not computed for boundary difference (the training is based on a single image)

Summary of training parameters and results

Run-Time Inspection

The table below summarizes the parameters that control run-time inspection and the main inspection results for each inspection mode.

Type	Inspection Mode	Item	Notes
Inspection parameters	Intensity difference	Method for affine-transforming the run-time region	The type of sampling to be used to affine transform the run-time region (see *Affine Transformation* on page 394)
		Method for normalizing run-time regions	The method used to perform grey-scale normalization on the run-time region (see *Normalizing Run-Time Images* on page 394)
	Feature difference	Match quality thresholds	Define the match quality interval to be used (see *Computing Feature Difference Information* on page 396)
		Minimum feature size	The smallest feature size to consider (see *Computing Feature Difference Information* on page 396)

Type	Inspection Mode	Item	Notes
		Minimum feature contrast	The smallest feature contrast to consider (see *Computing Feature Difference Information* on page 396)
		Boundary deformation	The smallest distance between feature boundary points to be considered for a match (see *Computing Feature Difference Information* on page 396)
		Ignore polarity	Controls whether to ignore polarity during inspection (see *Computing Feature Difference Information* on page 396)
	Blank scene inspection	Minimum feature size	The smallest feature size to consider (see *Localization of Defects* on page 404)
		Minimum feature contrast	The smallest feature contrast to consider (see *Localization of Defects* on page 404)
Inspection results	Intensity difference	Correlation score	Normalized correlation of pixel values between run-time region and trained region
		Thresholded difference region	Thresholded difference between template and run-time region
	Feature difference	List of matched features	Features present in trained region and present in run-time region
		List of missing features	Features present in trained region but missing from run-time region
		List of extra features	Features present in run-time image but not in trained region
	Blank scene inspection	List of feature defects	The extra features that disrupt the blankness of the run-time region

Summary of inspection parameters and main inspection results

BoundaryInspect

This chapter describes BoundaryInspect, a tool that uses a geometric model to inspect and judge the quality of a model likeness in target images. The chapter contains the following major sections:

Some Useful Definitions on page 407 defines some terms you will encounter as you read this chapter.

BoundaryInspect Tool Overview on page 408 describes the basic capabilities of the BoundaryInspect tool.

How The BoundaryInspect Tool Works on page 409 describes how the tool accomplishes boundary inspection.

Using the BoundaryInspect Tool on page 412 outlines the steps for using the BoundaryInspect tool.

Statistical Training on page 415 describes the Statistical Training tool you can use to create shape tolerances from training images.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

boundary: See **model boundary**.

capture range: A tolerance range used in statistical training. It specifies the range within which you expect to find contours in statistical training images. Contours outside the capture range are ignored.

client coordinates: A physical coordinate frame related to image coordinates by the *clientFromImage* transform.

clipping region: A closed convex polyline used in training to define the parts of the model where training is to take place.

contour: See **image contour**

featurelet: A common data structure used by Cognex vision tools that describes the sub-pixel location of an edge along with its angle, weight and magnitude.

featurelet chains: A series of featurelets that describe a contour extracted from an image.

image contour: Edges the BoundaryInspect tool finds in images that it attempts to match to model boundaries.

image coordinates: The coordinate frame used by images. An x-y space measured in pixels.

model boundary: The geometric description of the model to be inspected. In CVL, this description is done through **ccShape** derived classes.

model boundary tolerance: The tolerance of the model boundary when used in a boundary inspection. Points on the boundary (perimeter positions) have a positional tolerance and an angular tolerance. Image contours must match the model boundary within the boundary tolerance range.

model coordinates: The coordinate frame used to define a model.

perimeter position: A point on a shape perimeter or boundary.

perimeter range: A section of a shape perimeter represented by a perimeter start position and a signed range length. A perimeter range is a series of **perimeter position**s.

shape description: A geometric shape, described by a class derived from **ccShape**, representing the high-contrast boundaries of an object in an image. A shape description represents shape model properties (polarity and weight information) for each boundary it defines. These properties can be implicit (represented by a pure shape) or explicit (represented by a shape model). Also called a *geometric description*.

shape perimeter: Same as **model boundary**.

shape perimeter data: A data set where each data point is associated with a point on the shape perimeter. For example, if the data set is boundary tolerances, it specifies the allowed tolerance for the model boundary when used by the BoundaryInspect tool.

shape mask: A data set of mask values either *masked* or *unmasked*. Each mask value is associated with a perimeter position on the shape boundary. Unmasked positions are processed by the tool and masked positions are ignored.

shape tolerance: Same as **model boundary tolerance**.

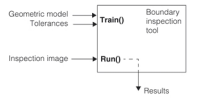
statistical training: An automated way of producing shape tolerances for a given shape description by using defect free training images. See the **ccShapeTolStats** class.

BoundaryInspect Tool Overview

Cognex vision tools allow you to analyze target images in various ways to look for specific patterns and shapes. Generally a tool locates a trained pattern in an image using parameters you can adjust to specify how close the target image matches the trained model. Even with very tight tolerances however, it is not possible to tell if you have an exact match, or to isolate the non-matching features.

The BoundaryInspect tool is designed to solve this problem. You train the tool with a geometric model which can be any of the **ccShape** derived classes in CVL. All of the lines that define the shape are termed *boundaries* and corresponding lines in inspection images are termed *contours*. When you train the tool you specify how closely found contours must match corresponding boundaries for a successful match. The boundaries can be divided into sections called perimeter ranges and each range can have different tolerances. Corresponding sections in inspection images are stored as featurelet chain sets. The results of the inspection include all of the matching boundaries and contours, as well as unmatched boundaries and contours. Since these perimeter ranges and featurelet chain sets contain specific information about matched and unmatched points, you can easily identify any inspection failures.

The figure below shows a simplified diagram of the Boundary Inspection tool.



Boundary Inspection tool

When you train the tool you provide a geometric model and corresponding tolerances for the shape. This defines for the tool, at each position along the shape boundary, how closely the inspection image contours must match the model. You have the choice of creating these tolerances manually, or generating the tolerances automatically using a statistical training tool provided by Cognex. The Statistical Training tool is described in the section *Statistical Training* on page 415.

Coordinate Systems

The BoundaryInspect tool uses three coordinate systems: the *image* coordinate system, the *client* coordinate system, and the *model* coordinate system. These are described in the figure below.

Coordinate systems

Image space is the x-y frame used for images and uses pixel units. The *clientFromImage* transform is used to transform image space into client coordinates, (typically a physical space) the BoundaryInspect tool uses for its calculations. The model used to train the tool is defined in model space and you provide a *pose* transform that relates model space to client coordinates. The tool requires that the model be registered with the image. That is, the transformed model into client space should be accurately registered with the inspection images so that the tool need not perform scaling or

rotation to inspect the image. The *pose* transform can be determined by an alignment tool such as PatMax or another method of your choosing.

Geometric Models

You train the BoundaryInspect tool with a geometric model (shape description) which can be any of the **ccShape** derived classes in CVL including shape trees which are hierarchical shape combinations. Models can also be obtained from CAD import (see the **ccCADFile** class), by using the *ModelMaker* tool, or by simply using vision tools to do edge extraction with chaining, and then converting the resulting edgelets to featurelet chains and then into polylines.

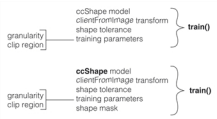
How The BoundaryInspect Tool Works

You prepare the BoundaryInspect tool by training it with a geometric model. The model can be any shape derived from the **ccShape** class including shape trees which are hierarchical shape combinations. (See the *Shapes* chapter of the *CVL User's Guide* for information about shapes).

Once trained the tool can be used repeatedly to inspect images. Each inspection produces a result object you can analyze to measure the inspection quality.

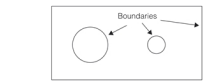
Training

You train the tool by calling its **train()** method. Two overloads are provided, one that includes a shape mask and one without. See the figure below.



Training

The model can be any shape derived from the **ccShape** class. A large variety of shapes are provided by CVL or you can derive you own shape class if necessary. A simple example of a training model is shown in the figure below.



Shape example

The rectangle and both circles all represent boundaries in this shape example.

The *clientFromImage* transform will be used at run time to transform target images to client coordinate space.

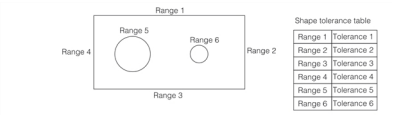
Shape Tolerance

When you run the BoundaryInspect tool it extracts contours from the inspection image and then attempts to match these contours with model boundaries. The matching is done on a point-by-point basis for points along the model boundary and corresponding points on the inspection image contours. Boundary points are represented by *perimeter positions* which are grouped into *perimeter ranges*. Image contour points are represented by *featurelets* which are grouped into *featurelet chain sets*.

Perimeter positions are compared to corresponding featurelets and a comparison is made of the *location* and the *tangent angle*. Each perimeter range is associated with a tolerance value for both location and angle and if the model boundary position and the image contour featurelet are within the tolerance range, they are considered a match. If not, they do not match.

Creating the tolerances for your shape description is a key part of tool training and is also key to creating an inspection tool that meets your needs. You can create shape tolerances manually or by any means you devise, but you must

produce a shape tolerance table of shape perimeter ranges with corresponding tolerances. Using the shape example in *Shape example* on page 409 we have divided the shape into six perimeter ranges. See the figure below:



Shape tolerance table

Tolerance 1 applies to Range 1, Tolerance 2 applies to Range 2, and so on. This scheme gives you great flexibility in assigning different tolerances to different parts of the model.

Global Tolerance

Cognex provides the templated class **ccShapePerimDataTable** that encapsulates the boundary ranges and associated tolerances. If you wish to use a global tolerance for all boundaries, you can leave the range and data vectors of that class empty and a default tolerance in the class will be used for all boundaries.

Statistical Training Tool

Cognex provides an automated approach for generating a shape tolerance table called *statistical training*. When you use statistical training you provide a number of training images that are defect free. The statistical training tool finds the contours in these training images and creates perimeter position *location* and *angle* tolerances based on the found data. The tolerance ranges set depend on the model type you are using; the normal distribution model, or the non-parametric model. The normal distribution model uses a *confidence level* parameter while the non-parametric model uses a *slack tolerance* parameter. These parameters are discussed in the **ccShapeTolStatsModelParams** reference page. The end result is a shape tolerance object you can use to train the BoundaryInspect tool.

More about statistical training is included in the section *Statistical Training* on page 415.

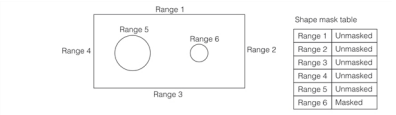
Training Parameters

The BoundaryInspect tool uses two training parameters, *granularity* and *clip region*. Granularity specifies a subsampling factor for subsampling images prior to inspection. When you subsample you then have fewer pixels to process and generally your inspections will run faster. However, as you raise the subsampling factor you lose resolution and your inspection accuracy may suffer. Finding the optimum subsampling factor is application dependent and may require some experimenting.

The clip region is a region you define in model coordinates that includes parts of the model you wish to train, and excludes those parts you do not wish to train. If there are model boundaries that are not important to your application you should exclude these boundaries with a clip region to decrease your inspection execution time.

Shape Mask

The shape mask is an optional mask you can apply to a trained model boundary where each perimeter range of a shape description can be masked or unmasked. If a range is masked, it is ignored by the inspection. Only unmasked ranges are used in the inspection. Using a shape mask allows you to fine tune your inspections to omit model boundary parts that are possibly troublesome and are not important to the inspection result. You must create a shape mask table similar to the shape tolerance table described in the *Shape Tolerance* on page 409. Using that same example the figure below illustrates a shape mask table that would mask out the smaller circle and eliminate it from the inspection.



Shape mask table

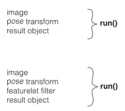
Global Mask

Cognex provides the templated class **ccShapePerimDataTable** that encapsulates the boundary ranges and associated mask values. If you wish to use a global mask value for all boundaries, you can leave the range and data vectors of that

class empty and a default mask value in the class will be used for all boundaries.

Inspection

You run an inspection by calling the **run()** method. Two overloads are provided, one that includes a featurelet filter and one without. See the figure below.



Inspection

When you run the BoundaryInspect tool it smooths and subsamples the input image as specified by the granularity provided in the training parameters. The image contours are then extracted and converted to featurelet chains and then transformed to model coordinate space. If you chose the **run()** method that specifies a featurelet filter, the extracted featurelets are then filtered by the filter you provide.

The BoundaryInspect tool then proceeds with matching the model boundary with the remaining image contours (featurelet chains) to produce matched and unmatched model boundary and image contours. Matching is done with the unmasked boundary only. An image contour (partial or whole) is considered to match a model boundary (partial or whole), if the distance between the contour elements (featurelets) and the model boundary is within a *distance* tolerance range. In addition, each contour element should have at least one boundary point in its distance tolerance with a normal angle whose difference with the contour element gradient angle is not more than *angle* tolerance. All such boundary to contour matches will be located and reported without violating the following rules that define the matching details and ambiguity resolution criteria:

- No model boundary can match more than one image contour. An image contour, on the other hand, can match more than one model boundary or it can match the same model boundary more than once at different non-overlapping segments. This rule applies to both partial and whole model boundaries and contours.
- The shorter image contour has the priority if two image contours match the same whole or partial model boundary.
- If two image contours match the same model boundary at two overlapping boundary segments, the one that has the longer perimeter matching model boundary is chosen as the match.
- Image contours that have closer start and end points to the model boundaries are favored.

You can specify different boundary tolerances for different sections of the model boundary. The matching process uses the local boundary tolerances while deciding if an image contour element matches a local boundary section.

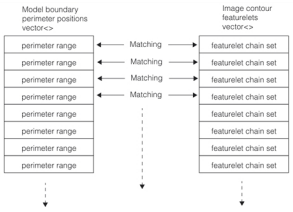
Note that model boundaries are not allowed to be self intersecting.

Results

The inspection attempts to match model boundaries with inspection image contours. Model boundaries are expressed as perimeter ranges, a series of perimeter positions along a boundary. Image contours are expressed as featurelet chain sets.

Results are either matched or unmatched. The result object holds all of the unmatched model boundary perimeter positions as a vector of perimeter ranges. All of the unmatched image contour featurelets are stored as one featurelet chain set.

Matched image contour featurelets are stored as a vector of featurelet chain sets and matched model boundary perimeter positions are stored as a vector of perimeter ranges. Each perimeter range corresponds to a matching featurelet chain set. See the figure below.

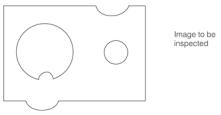


Matching results

The figure below is an example of a boundary model along with the boundary tolerances shown as the shaded area. If matching image contours lie within the shaded area they are considered a match. Note, that to make our point in this example, the inside tolerances are smaller than the outside tolerances.

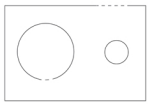
Example boundary model

The figure below is an example image to be run by the BoundaryInspect tool trained with the model above. For this example we have purposely deformed the image to include features that fall within the tolerance range, and features that are outside the tolerance range.



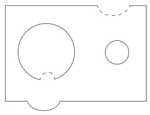
Example image

The results of this inspection are shown in the next two figures below.



Matched and unmatched model boundaries

In the figure above the matched boundaries are shown as solid lines and the unmatched boundaries are show as dotted lines. Unmatched boundaries occur where the image contours were outside the tolerance range.



Matched and unmatched image contours

In the figure above the matched image contours are shown as solid lines and the unmatched contours are show as dotted lines. Unmatched contours occur where they were outside the tolerance range.

Using the BoundaryInspect Tool

To use the BoundaryInspect tool you need to work with the CVL classes summarized below.

Class	Description
ccBoundaryInspector	The BoundaryInspect tool. Contains the **train()** and **run()** functions.
ccBoundaryInspectorTrainParams	The BoundaryInspect tool training parameters.

Class	Description
ccBoundaryInspectorResult	The BoundaryInspect tool results.
ccShapeTol	Associates tolerance data with CVL shapes.
ccShapeTolTable	
ccShapeMask	Associates mask data with CVL shapes.
ccShapeMaskTable	
ccShapeMaskValue	Mask values for **ccShapeMask**.
ccBoundaryTol	Defines acceptable tolerances for model perimeter positions and image contour featurelets. Tolerance data for **ccShapeTol**.

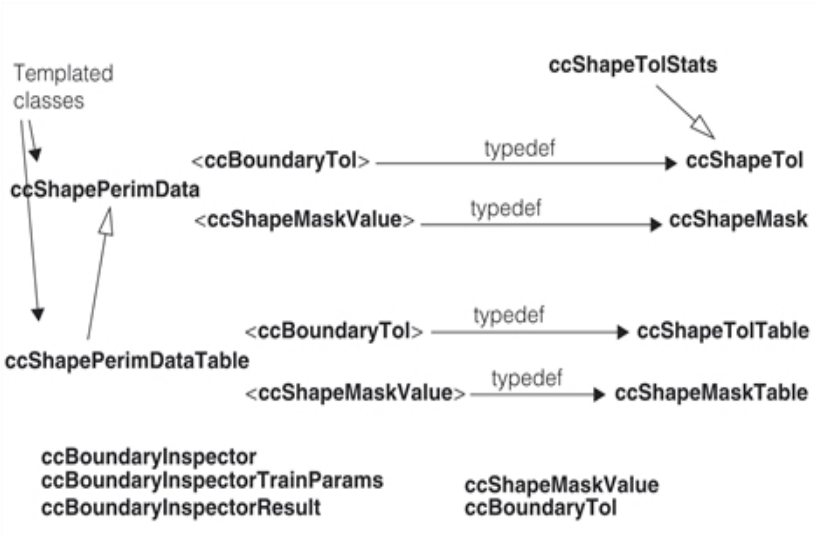
BoundaryInspect tool classes

The following classes are required for statistical training. See *Statistical Training* on page 415.

Class	Description
ccShapeTolStats	The statistical tolerance training tool.
ccShapeTolStatsParams	The statistical tolerance training tool parameters.
ccShapeTolStatsModelParams	The statistical tolerance training tool model parameters.

Statistical training classes

These classes are related as shown in the following class hierarchy:



Class hierarchy

Note that **ccShapeTolTable** is not explicitly derived from **ccShapeTol**. Instead the derivation is implied by the template instantiation of the **ccShapePerimDataTable** class. However, in your code you can safely assume that **ccShapeTolTable** is derived from **ccShapeTol**.

The same applies to **ccShapeMaskTable** and **ccShapeMask**.

To perform a boundary inspection you must first instantiate a **ccBoundaryInspector** tool. You then need to train the tool before running inspections. These steps are covered in the following sections.

Training

To train a BoundaryInspect tool you call the **train()** method. There are two overloads, one with a shape mask and one without a shape mask. See below.

```
void train(
 const ccShape &modelBoundary,
```

```
 const cc2XformBase &clientFromImage,
 const ccShapeTol &shapeTolerance,
 const ccBoundaryInspectorTrainParams &trainParams,
 ccDiagObject* obj = 0,
 c_UInt32 diagFlags = 0);

void train(
 const ccShape &modelBoundary,
 const ccShapeMask &shapeMask,
 const cc2XformBase &clientFromImage,
 const ccShapeTol &shapeTolerance,
 const ccBoundaryInspectorTrainParams &trainParams,
 ccDiagObject* obj = 0,
 c_UInt32 diagFlags = 0);
```

The training arguments are discussed below:

ccShape
Your training model must be a class derived from **ccShape**. CVL provides an extensive library of shape descriptions or you can create your own **ccShape** derived shape description.

ccShapeMask
If you use a shape mask you should pass in a **ccShapeMask** reference here. You will need to create boundary perimeter ranges for your shape description and assign mask values to the ranges. See **ccShapePerimDataTable**.

cc2XformBase
Pass in the transform you intend to use with inspection images.

ccShapeTol
Pass in a **ccShapeTol** reference here. This can be a reference to a **ccShapeTable** or a reference to a **ccShapeTolStats**. You will need to partition your model into perimeter ranges, and then assign tolerances (**ccBoundaryTol** objects) to each range. See **ccShapePerimDataTable**.

ccBoundaryInspectorTrainParams
If you are working with good quality inspection images you may wish to specify a granularity greater than 1.0 to cause subsampling which may reduce your inspection time. Some testing may be required to find the optimum granularity for your application.

If you wish to use only part of the training model for inspections, specify the clip region as a closed convex **ccPolyline** in client coordinates. Only the model boundaries enclosed by the clip region will be used for inspections.

Inspection

To run a BoundaryInspect tool inspection you call the **run()** method. There are two overloads, one with a feature filter and one without. See below.

```
void run(
 const ccPelBuffer_const<c_UInt8> &image,
 const cc2XformBase &pose,
 ccBoundaryInspectorResult &results,
 ccDiagObject* obj = 0,
 c_UInt32 diagFlags = 0) const;

void run(
 const ccPelBuffer_const<c_UInt8> &image,
 const cc2XformBase &pose,
 const ccFeatureletFilter &featureletFilter,
```

```
ccBoundaryInspectorResult &results,
ccDiagObject* obj = 0,
c_UInt32 diagFlags = 0) const;
```

The inspection arguments are discussed below:

ccPelBuffer_const
The inspection image.

cc2XformBase
The pose transform transforms the model into client coordinates. You must set the transform so that the model is perfectly registered with the expected image. The BoundaryInspect tool does no scale or rotation adjustments during the inspection.

ccFeatureletFilter
If your inspection images yield extraneous featurelets that are not pertinent to the inspection, you may wish to eliminate some of these featurelets before the inspection begins. The featurelet filter is applied in model coordinates.

Include a featurelet filter here if it benefits your application.

ccBoundaryInspectorResult
Provide a result object where the tool will place the inspection results.

Results

The result object contains all of the matching boundaries and contours, as well as those that do not match. If there are no non-matching boundaries or contours you have a perfect inspection. However, this is seldom the case. Of interest is the **unmatchedModelBoundary()** function which returns perimeter ranges in the model that found no match in the image. Also **extraImageContours()** which returns all of the extracted featurelets that did not match model boundaries. You will need to design code into you application to analyze these results to help you understand and locate inspection failures.

Diagnostics

When you train the BoundaryInspect tool and also when you run inspections you provide a diagnostic object and diagnostic flags. These are standard CVL vision tool diagnostics you can use to record tool inputs, intermediate results, and final results. After completing training or an inspection you can display the recorded diagnostics to trouble shoot failures. The diagnostics provide a graphical display that can be helpful in understanding how the tool is performing. A detailed discussion of how to use CVL diagnostics is included in the chapter *PatMax* on page 275.

Guidelines

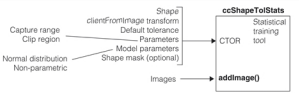
To obtain optimum performance from the BoundaryInspect tool use the following guidelines:

1. The absolute scale components of the *pose* should be close to 1.0 for best results. Large deviations of scale from 1.0 in combination with tight angle and distance tolerances will degrade the tool performance.
2. The BoundaryInspect tool performs well using nonlinear transforms. However, tool performance degrades if local scale values of the transform change greatly from image corner to image corner.
3. To make your application run faster use the largest granularities that produce good results.

Statistical Training

Statistical Training is a tool you can use to automatically create boundary tolerance information from sample images. You create the tool with the parameters as shown in *Statistical training tool* on page 416 below. You then train the tool by providing sample images through calls to **addImage()**. Once you have trained the tool with these images you pass it to the BoundaryInspect tool **train()** function as the shape tolerance argument.

The Statistical Training tool uses the provided shape description and the training images to establish an empirical tolerance range around the shape boundaries that will allow the BoundaryInspect tool to accept all of the training image contours.



Statistical training tool

Statistical tool parameters are discussed in the following paragraphs.

You create a statistical training tool using the shape description and *clientFromImage* transform you are using in your boundary inspection application. You provide a clipping region which encloses all parts of the shape boundary where you want statistical tolerances to be calculated. You can specify a capture range which can exclude parts of the training images not required by your application and may save you processing time.

The model parameter specifies how the tool should interpret the statistical data. You can specify either *normal* distribution or *non-parametric* distribution. The first method assumes that the observed positional and angular differences between the model boundary and the image contours can be modeled by normal distribution. The tolerances are decided on using standard statistical inference methods and a confidence level you provide. This method requires at least two observations for a boundary position to be able to estimate the tolerance for that position. If normal distribution applies, this method has several advantages such as taking a number of training images into account and having the capability to use a confidence level.

The second method does not assume normal distribution or any other parameterized statistical model. The model boundary tolerances are estimated by taking the largest observed distance and angle differences, and adding the *slack tolerances* you provide to these differences. This method requires at least one observation for a boundary position to be able to estimate the tolerance for that position. The non-parametric method is useful if you are not confident about the distribution of the image contour positions and angles.

When you create the tool you also specify a default tolerance which is used as the tolerance when there is not enough statistical information to compute a statistical tolerance.

The shape mask is an optional mask you can apply to the model boundary where each position on the boundary can be masked or unmasked. If a position is masked, it is ignored by the tool. Only unmasked position are used in statistical calculations.

To create a statistical data base for tolerance calculations you add images to the tool, one at a time. These should be non-defective images that are good quality examples of target images in your application. An overload to **addImage()** allows you to apply a featurelet filter to the contours extracted from the added image. If you are using a featurelet filter in your boundary inspection application you may wish to use the same filter here for consistency.

ID Tool

The ID tool locates and decodes one or more symbols in an input image. It can also characterize the print quality of each symbol. The symbols within the input image can be of different symbologies and can also be arbitrarily positioned and oriented. The ID tool does not require any training.

Note: The ID tool provides similar functionality to the Barcode tool. The ID tool provides better performance, can decode more types of symbols, and has a much simpler programming interface. Cognex recommends that you use the ID tool for all new application development.

This chapter includes the following sections:

Some Useful Definitions on page 417 defines some terms used in this chapter.

ID Tool Overview on page 417 provides an overview of the ID tool.

Using the ID Tool on page 419 describes how to use the ID tool.

Some Useful Definitions

element: An individual bar or space from a symbol.

quiet zone: The area surrounding a symbol that may not contain any marks.

symbol: A collection of bars and spaces that encode data.

symbology: How a symbol encodes data. Symbologies specify the size, shape, order, and meaning of the elements that make up the symbol.

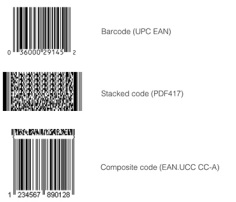
ID Tool Overview

The ID tool locates and decodes one-dimensional (1-D) symbols. A 1-D symbol is made up of a set of parallel lines and spaces used to represent data.

The ID tool can find and decode three types of symbols:

- Barcodes, which are made up of bars and spaces of varying width.
- Stacked codes, which consist of multiple stacked barcodes
- Composite codes, which are made up of barcodes and non-1-D symbols.

Examples of each symbol type are shown in the figure below.



Symbols

The ID tool can locate and decode one ore more symbols in a single input image. If the tool is configured to locate and decode multiple symbols, the symbols may be of different types and symbologies.

In addition to decoding symbols, the ID tool can also compute and report standard measures of symbol quality.

ID Tool Operating Modes

The ID tool supports the following two operating modes:

- Standard

  Use the standard operating mode in easy-to-read cases. The run-time speed in this mode is more predictable than in the 1DMax mode. This is the default mode.

- 1DMax

  Use the 1DMax mode primariliy in hard-to-read cases; however, you can use it in easy-to-read cases as well. In this mode, the run-time speed depends more heavily on the image content. In particular, the run time may exhibit significantly larger variations when the number of symbols to find specified by **ccIDSearchParams::numToFind()** is greater than what can be decoded from the scene. You can use the **ccTimeout** object (specified in *ch_cvl/attent.h*) to gain better control over the tool's run time.

Supported Symbologies

The table below lists the symbology types supported by the ID Tool.

Symbology	Characteristics
Interleaved 2 of 5	Popular in warehouse applications, Interleaved 2 of 5 (also called I-2/5 or ITF) is a variable-length, numeric-only code. A symbol using Interleaved 2 of 5 symbology encodes 2 characters in a unit of 5 bars and spaces, where the even position character is encoded into bars while the odd position character is encoded into spaces. The Interleaved 2 of 5 symbology can only encode data with an even number length. The symbol can include a checksum character for error detection.
Code 39	Code 39 (also called USS Code 39 or Code three of nine) is a widely-used symbology developed for use in non-retail environments, and can encode both letters, digits, and special characters such as " % " and " / ". A symbol using Code 39 symbology encodes each character using 5 bars and 4 spaces for a total of 9 elements, and 3 out of the 9 elements are always wide. The symbol can include a checksum character for error detection.
Code 128	Code 128 is a very high-density alphanumeric symbology which can be scanned bidirectionally. The symbology can encode the entire 128 ASCII character set plus four non-data characters. A symbol using Code 128 symbology encodes each character using 11 black or white modules, and each symbol includes a checksum character.
UPC/EAN	UPC is a fixed-length, numeric-only symbology which can be scanned bidirectionally. The size for a UPC symbol can vary to accommodate various printing processes, but the code works best when the height of the symbol exceeds its width. European companies use the generally equivalent European Article Numbering (EAN) system.
4-State	4-State is an alphanumeric symbology adopted by the Australian Post. A barcode using the 4-State symbology encodes each character using 4 different types of bars, each of which has a distinct name and value. A 4-State barcode can be generated in one of three different structures of 37 bars (standard), 52 bars, or 67 bars. The ID tool supports the Australian, UPU, Japan Post and USPS versions of the 4-state symbology.
POSTNET	The Postal Numeric Encoding Technique (POSTNET) barcode was invented by the US Postal Office to encode ZIP Code information. A barcode using the POSTNET symbology encodes each numeric character using a combination of 5 bars, either short or long. A POSTNET barcode can contain a 5-digit ZIP Code, a 5-digit ZIP + 4 Code, or an 11-digit delivery point code. The symbol always includes a checksum character.
PLANET	The PLANET barcode is the inverse of the POSTNET barcode, using short bars where the POSTNET symbology uses long bars and long bars where the POSTNET symbology uses short bars. The US Postal Office uses PLANET barcodes to track pieces of mail. A PLANET barcode can have up to 12 digits.

Symbology	Characteristics
CEPNet	CEPNet is a Brazilian postal barcode. CEPNet and POSTNET use the same symbol character set and checksum algorithm but the lengths are different. A CEPNet barcode can contain an 8-digit CEPNet code, an 8-digit CEPNet code + 1 checksum character, a 13-digit CEPNet code, and a 13-digit CEPNet code + 1 checksum character.
RSS	Reduced Space Symbology (RSS) was developed as a family of seven linear symbologies to provide users with features that address specific space limitation and application needs. RSS is designed to allow encoding of up to 74 characters of data. RSS allows greater capacity for encoding and capturing data such as product supplier, expanded product identification, price per pound, extended price, product weight, and so on. The ID tool supports a variety of RSS types.
Code 93	Code 93 is an alphanumeric code similar to Code 39 and can encode 48 different characters. Code 93 is more compact than Code 39, but is not as widely used.
Codabar	Developed in the early 1970s, Codabar is designed to be accurately read even when printed on dot-matrix printers for multi-part forms such as FedEx Airbills and blood bank forms. Although newer symbologies hold more information in a smaller space, Codabar has a large installed base in libraries.
Pharmacode	Also known as Pharmaceutical Binary Code, Pharmacode is used in the pharmaceutical industry as a packing control system. It is designed to be readable despite printing errors.
PDF417	PDF417 is a stacked linear bar code symbol used in a variety of applications, including transport, identification cards, and inventory management.

Supported symbologies

Composite Symbols

A composite symbol is formed by combining a barcode with a second symbol, usually a high-density stacked or 2-D symbol, as shown in the figure below.

Composite symbol

For the composite symbol type that it supports, the ID tool returns a single result that contains the decoded data from both parts of the composite symbol.

Using the ID Tool

This section provides information about how to use the ID tool.

Image Requirements

Images that you supply to the ID tool must meet the following requirements to qualify for the easy-to-read category:

- All symbols must be greater than 50 pixels in length, while the maximum width of any module, where a module is the narrowest element of the symbol (either a space or a bar), cannot exceed 10 pixels.
- For linear symbologies (where modules have different widths but uniform heights), a module must be at least 1.6 pixels wide and 50 pixels high. For postal codes (where the modules are of uniform width but varying height), a module must be at least 2.5 pixels in width.

- A quiet zone (an area on either end of a symbol with no marks) of at least the minimum size specified in the print specification for the barcode must be present.
- The contrast between modules and background must be at least 32 grey levels.
- The pixel aspect ratio can be no greater than 1.35 to 1.

Images that fail to meet these requirements are considered hard-to-read images.

The following figure shows an example image of a POSTNET code that meets these requirements.

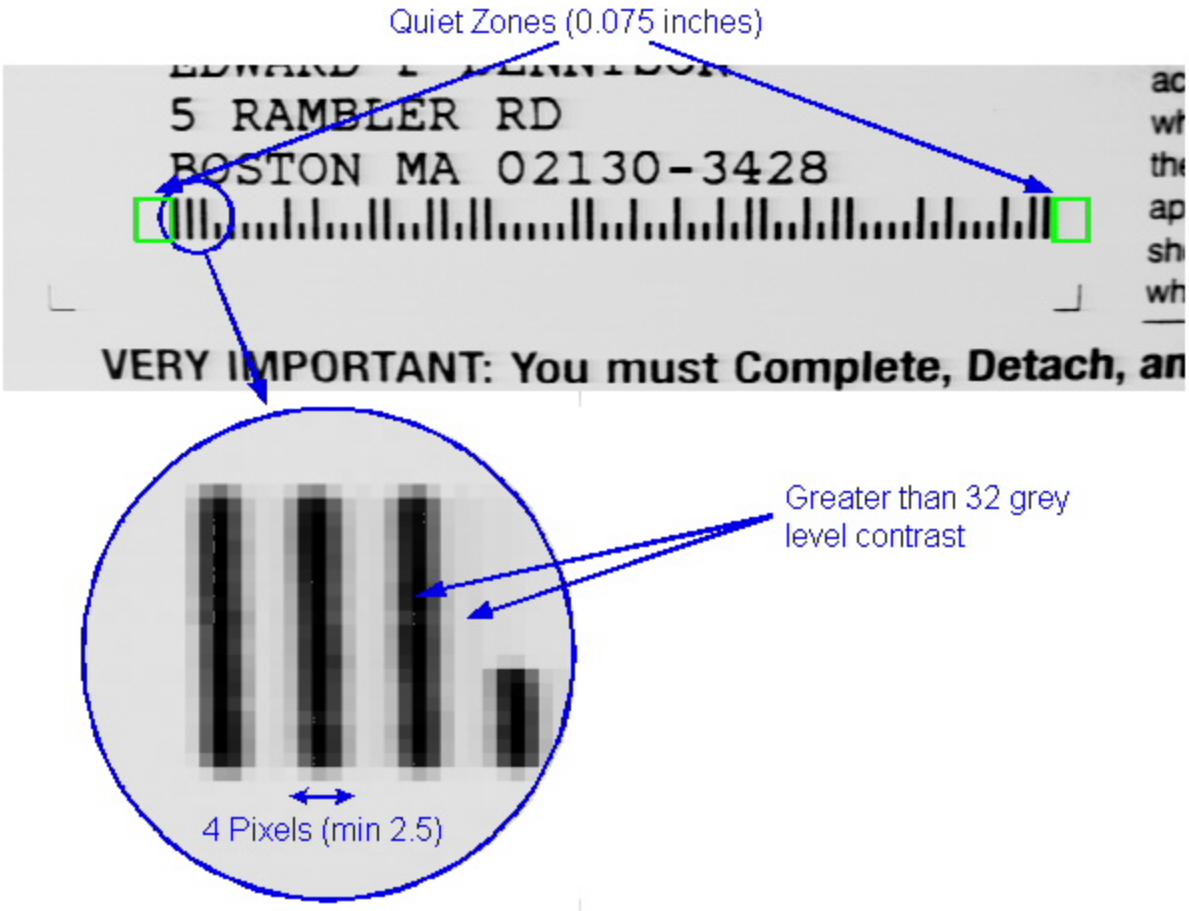


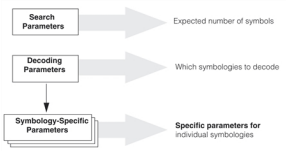
Image requirements (POSTNET)

Note: The specific requirements for quiet zone are different for different symbol types. Refer to the specifications for those symbologies for their quiet zone requirements.

ID Tool Parameters

The ID tool accepts two parameter objects as input when it decodes a symbol. The search parameters describe the expected number and appearance of the symbol or symbols in the image while the decoding parameters specify which symbology types can be decoded. A separate parameter object for each symbology type is used to specify symbology-specific parameters.

The figure below shows how the parameters are organized.



ID Tool parameters

Search Parameters

The ID tool accepts the search parameters listed in the table below.

Result	Description
Number to find	The ID tool will only locate and attempt to decode the number of symbols that you specify.
Mirroring	Whether or not the symbols are mirrored (reflected about the image Y-axis) in the image. If you specify multiple symbols, they all must have the same mirroring.
Polarity	Whether the symbol consists of dark bars on a light background or light symbols on a dark background. If you specify multiple symbols, they all must have the same polarity.
Omnidirectionality (for postal codes only)	Whether the ID tool locates and decodes postal codes placed at arbitrary orientations (omnidirectional search) or only those placed at approximately horizontal orientation.
Operating mode	Whether the tool should use the 1DMax or the Standard mode.

Search parameters

Decoding Parameters

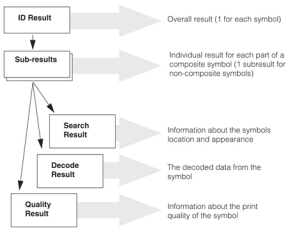
The ID tool accepts the decoding parameters listed in the table below.

Result	Description
Enabled symbologies	Which symbologies to attempt to find.
Quality option	The quality measures to compute.

Decoding parameters

ID Tool Results

Each time you run the ID tool on an input image, it produces an aggregate result that provides information about all of the symbols found in the input image. The figure below shows how the results are organized.



ID Tool results

Each of the components of the results are discussed in this section.

Search Results

The ID tool provides the information listed in the table below for each found symbol.

Result	Description
Angle and location	The location and orientation of the found symbol in the client coordinates of the input image.
Region	A polyline that traverses the four corners of the symbol.
Module size	The symbol module size, in client units.
Mirroring	Whether or not the found symbol was mirrored.
Polarity	The polarity of the found symbol.

Search results

Decode Results

The ID tool provides detailed information about the decoded data from the symbol. The decode results are summarized in the table below. Because the tool can locate and decode multiple symbols with different symbologies in the same image, the decode results include information that lets you identify the symbol's type and other characteristics.

Results	Description
Symbology Identifiers	A three-character ASCII string that encodes information about the symbol, as defined by ***ISO/IEC 15424:2000*** *Information technology -- Automatic identification and data capture techniques - Data Carrier Identifiers (including Symbology Identifiers)*.
Decoded string	The decoded string is available as a string, as a an array of multibyte characters, and as an uninterpreted (raw) stream of data.
Symbology	The symbology of the decoded symbol.
Symbology subtype	The tool records the actual type of this symbol, which is valid for the UPC/EAN, RSS and 4-State symbologies.

Decode results

Quality Results

In addition to information about where the symbol was found and what the decoded data from the symbol, the ID tool can also return information about the print quality of the symbol. Not all quality measures are available for all symbols, and you can specify which measures to compute.

The table below lists the quality measures that the ID tool can compute. The **Symbologies** column lists the symbologies for which the measure is available. In all cases, the quality measure is reported as one of five letter grades from A to F, where A is excellent and F is failed.

Measure	Description	Symbologies
Decodability	Decodability is the measure of the accuracy of the printed bar code against the appropriate reference decode algorithm. Each symbology has published dimensions for element widths and provide margins or tolerances for errors in the printing and reading process. Decodability measures the amount of margin left for the reading process after printing the bar code.	2 of 5 Code 39 Code 128 UPC/EAN RSS Code 93 Pharmacode PDF417 Composite
Defects	Defects are voids found in bars or spots found in the spaces and quiet zones of the code. Each element is individually evaluated for its reflectance non-uniformity. Element reflectance non-uniformity is the difference between the highest reflectance value and the lowest reflectance value found within a given element.	2 of 5 Code 39 Code 128 UPC/EAN RSS Code 93 Pharmacode PDF417 Composite

Measure	Description	Symbologies
Minimum Edge Contrast	Each transition from a bar to a space, or back again, is an edge whose contrast is determined as the difference between peak values in that space and that bar. Each edge in the scan profile is measured, and the edge that has the minimum contrast from the transition from space reflectance to bar reflectance, or from bar to space, is the Minimum Edge Contrast.	2 of 5 Code 39 Code 128 UPC/EAN RSS Code 93 Pharmacode PDF417 Composite
Edge Determination	In order to discern bars and spaces, a Global Threshold is established on the scan reflectance profile by drawing a horizontal line half way between the highest reflectance value and the lowest reflectance value seen in the profile. Edge Determination can then be done by counting the number of crossings at the Global Threshold confirming whether the count conforms to or is considered non-conforming to a legitimate barcode symbology. If the barcode conforms it passes and gets a grade of A, if it fails it gets a grade of F.	2 of 5 Code 39 Code 128 UPC/EAN RSS Code 93
Modulation	Modulation has to do with how a scanner views wide elements (bars or spaces) in relationship to narrow elements, as represented by reflectance values in the scan profile. Scanners usually view spaces narrower than bars and scanners typically view narrow spaces being even less intense or not as reflective as wide spaces.	2 of 5 Code 39 Code 128 UPC/EAN RSS Code 93 Pharmacode PDF417 Composite
Reference Decode	The tool gives the symbol a grade of A if the ANSI Reference Decode algorithm can decode this symbol, or graded F otherwise.	2 of 5 Code 39 Code 128 UPC/EAN RSS Code 93 Pharmacode PDF417 Composite
Scan Grade	The tool records the lowest grade received for any reflectance scan profile based quality measurement.	2 of 5 Code 39 Code 128 UPC/EAN RSS Code 93 Pharmacode PDF417 Composite
Symbol Contrast	Symbol Contrast is the difference between the highest reflectance value and the lowest reflectance value in the scan profile. The higher the value, the better the grade.	2 of 5 Code 39 Code 128 UPC/EAN RSS Code 93 Pharmacode PDF417 Composite
Overall Grade	The overall grade for the symbol, based on all supported quality measures.	All

Quality measures

Optimizing Performance

To obtain the best performance from the tool, observe these guidelines:

- Enable only the Barcode symbologies that you expect to be present in the acquired image; enabling unnecessary symbologies can negatively impact the performance of a ID tool.
- Be aware that the Pharmacode symbology is not omnidirectional, so the orientation of the symbol will affect the decoded result.
- RSS symbols should be aligned along the x-axis of the image for best performance.
- The ID tool is more robust (higher yield or decode rate) when decoding Postal Codes in non-omnidirectional mode, which is appropriate for images where the postal code is nominally oriented horizontally. Cognex suggests using non-omnidirectional mode for production systems decoding postal symbols when it is possible.
- The ID tool may perform poorly if you configure it to locate and decode multiple symbologies with similar characteristics. For example, Interleaved 2 of 5 symbols may be difficult to distinguish from Code 39, Code 128, or UPC/EAN symbols without additional constraints such as a difference in expected string length. The more information that you can supply about the expected symbols, the better the tool will perform.

Limitations

For best results, keep the following limitations in mind as you develop your ID tool application:

- The tool does not support multiple simultaneous sets of parameters for the same symbology. For example, you cannot use the tool to decode Interleaved 2 of 5 symbols both with and without checksum characters in the same image at the same time.
- The smallest value the Pharmacode symbology can decode is 31.

acuRead

acuRead is a vision tool for optical character recognition (OCR). Using acuRead, you can develop applications for reading character strings from silicon wafers or other surfaces.

This chapter provides the conceptual background for using acuRead on both CVL and OMI platforms. It discusses the following topics:

Some Useful Definitions on page 425 is a glossary of relevant terms.

acuRead Overview on page 426 describes acuRead's major features and applications.

Standard OCR on page 427 describes acuRead's general operation.

Using acuRead on page 431 describes the techniques that you use to implement an acuRead application.

Reading Special Characters with OCR on page 433 describes how acuRead reads special characters.

Framework Differences on page 443 describes differences in the acuRead features available in the OMI and CVL frameworks.

Except for noting feature differences between frameworks, this chapter does not describe how to run acuRead in OMI and CVL. For framework-specific details, see the following documents:

- *OMI Object Reference*, which describes the OMI Read object and its Font child
- *OMI User's Guide*, which describes how to use OMI
- *CVL Class Reference*, which describes the classes related to acuRead
- *CVL User's Guide*, which describes how to use CVL

Some Useful Definitions

brightfield lighting: Lighting that reflects light from the surface into the lens, causing scribed characters to appear dark on a bright background.

character acceptance threshold: The minimum score for a character to be considered found. acuRead computes this threshold based on the *string acceptance threshold*. When a character fails this threshold acuRead substitutes a * for the missing character. Also see *minimum character acceptance threshold*.

character score: A result value that measures the similarity between the characters in a field and the best match in the font model.

darkfield lighting: Lighting that reflects light from the surface away from the lens, highlighting the character strokes on a dark background.

field map: A map that defines a string acuRead will read. The map contains field positions, one for each character position in the string to be read. Each field position contains a field string that specifies the valid ASCII characters for that position.

field position: A character position within a string. Sometimes called a field.

field string: The valid characters for a given field position.

fixed width fonts: Fonts where all characters are the same width.

kerning: Reducing the spacing between characters to improve visual appearance.

light intensity: A tunable value controlling the overall brightness of the lights.

light mode: A tunable value controlling the balance of brightfield and darkfield lighting, which vary in an inverse relationship.

light tuning: An automatic operation that tests brightfield and darkfield light intensity values, sweeping through specified ranges by specified increments to find the best values.

line error: The maximum perpendicular distance, in pixels, from an ideal baseline fitted to the characters in the string. See *Setting the Line Error Tolerance* on page 436.

minimum character acceptance threshold: The minimum score for a character to be considered found that is a lower threshold than *character acceptance threshold*. acuRead computes this threshold based on the *string acceptance threshold*. When a character fails this threshold acuRead substitutes a ? for the missing character.

Non-Linear OCR™: A Cognex OCR technique that uses a permissive scoring algorithm.

process tuning: An automatic operation that finds the best combination of the enabled filters.

proportional fonts: Fonts where the font width varies depending on the character.

scale tuning: An automatic operation that adjusts the expected character height and width to the values that yield the highest string score.

standard OCR: An OCR technique that uses a restrictive scoring algorithm.

string acceptance threshold: The minimum score for a character string to be considered found.

string score: The composite score for the string computed from the individual character scores.

space error: The maximum deviation, in pixels parallel to the baseline, from equal character spacing.

Virtual Checksum™: A Cognex OCR technique compares the result strings from Standard and Non-Linear OCR. The checksum “passes” only if the result strings match.

acuRead Overview

acuRead performs optical character recognition (OCR). Its primary application is reading characters scribed on silicon wafers. The tool supports several predefined fonts as well as any user-defined font that meets our specifications. (See *User-defined Font Specifications* on page 440).

The tool supports two methods of reading characters:

- Standard OCR, which allows a narrow margin of error
- Cognex Non-Linear OCR, which allows for a wider margin of error (for predefined fonts only)

acuRead will run on a CVL platform or an OMI platform. However, the OMI platform does not support user-defined fonts.

acuRead tool capability is best thought of in terms of font support, either predefined fonts or user-defined fonts. The table below summarizes these capabilities:

	Predefined fonts	User-defined fonts
CVL platforms	Yes	Yes
OMI platforms	Yes	No
Fixed width fonts	Yes	Yes
Proportional fonts	No	Yes
Standard OCR	Yes	Yes
Non-Linear OCR	Yes	No
Variable length strings (no checksum)	Yes	Yes

acuRead tool summary

The following sections discuss the two methods of reading characters, and the acuRead tool features.

Standard OCR

Standard OCR reads characters by determining which of a set of possible characters is in each field, or character position, of a string. Once the tool has read the string, it returns the string along with additional data generated during the decoding operation.

When it decodes a string, the tool performs the following steps:

1. After an image is acquired, it can use one or more filters to preprocess the region of interest.
2. It locates the string and its individual fields.
3. It compares the character in each field to each character in the font model, computes a score for each comparison, and selects the highest-scoring character.
4. For each field, it compares the best character score to the character acceptance threshold for Standard OCR and passes that character if its score is greater than or equal to the threshold. A field whose score falls below the character acceptance threshold is a *failed field.* and the character is not found.
5. It computes the string score and builds the result string. It marks failed fields with either:
    - A “*” to indicate that the score for the character is below the character acceptance threshold, but above the minimum character acceptance threshold
    - A “?” to indicate that the score for the character is below the minimum character acceptance threshold
6. If the checksum is SEMI, BC412, or IBM412, it computes the checksum.
7. It reports the following results:
    - Character scores
    - Character locations
    - Number of fields
    - String score
    - Result string

Non-Linear OCR

When configured to use Non-Linear OCR, acuRead attempts to read the string twice. First, it performs a Standard OCR step, but does not report the results. Then it performs Non-Linear OCR using the string- and field-location information produced by the Standard OCR attempt, and a different acceptance threshold. It uses the *acceptNL* run-time parameter to compute nonlinear values for the character acceptance threshold and the minimum character acceptance threshold. It reports the following results:

1. The result string obtained using the Non-Linear OCR acceptance threshold, marking failed fields with either:
    - A “*” to indicate that the score for the character is below the character acceptance threshold, but above the minimum character acceptance threshold
    - A “?” to indicate that the score for the character is below the minimum character acceptance threshold
2. The string score obtained using the Non-Linear OCR acceptance threshold

    With Non-Linear OCR, OMI combines the checksum result with the string score. For more information about how acuRead calculates the score for OMI, see the section *String Scoring* on page 443.
3. The following additional data:

- Character scores
- Character locations
- Number of fields

Note that Non-Linear OCR cannot be used with user-defined fonts.

Tool Features

The major features of acuRead include:

- Filters for preprocessing the region of interest. Available filters are low pass, high pass, top hat, and clipping.
- Support for reading SEMI, Canon, OCR-A, and Triple fonts
- Support for reading strings created using user-defined fonts. User-defined fonts can be either fixed width or proportional width fonts.
- The ability to read fixed length and variable length strings
- The ability to set light mode and light intensity values, which together control the balance between brightfield and darkfield lighting
- Tuning of lights, filters, and character size
- Standard SEMI, BC412, and IBM412 checksums plus Cognex Virtual Checksum
- For each field, the ability to limit the candidate character set to any subset of alphanumeric characters
- Ability to read multiple special characters
- The following user-defined tolerances:
  - Minimum character score (the acceptance threshold)
  - Line error (deviation perpendicular to the baseline)
  - Space error (deviation parallel to the baseline)

Filters

acuRead can apply the following filters alone or in combination to preprocess the image:

- Clipping filter
- High pass filter
- Low pass filter
- Top hat filter

Clipping Filter

The clipping filter limits grey-scale levels to a particular range by changing all values above a maximum value to that maximum, and all values below a minimum value to that minimum. You can use this filter to eliminate very dark and very light anomalies in the image.

High Pass Filter

The high pass filter is a neighborhood-averaging difference filter that strips the image of low-frequency information. It helps eliminate patterns that shift from the background color to the foreground color in a distance greater than twice the thickness of a character stroke. It helps eliminate low-frequency noise (grey-scale intensity variations across the image such as long horizontal lines through or near the string). It also reduces blurring caused by gradual intensity variations.

Low Pass Filter

The low pass filter is a neighborhood averaging filter that strips the image of high-frequency information. In this context, high frequency refers to grey-scale patterns in the image with minimum dimensions less than half the thickness of a character stroke. This filter helps eliminate streaks and scratches.

Top Hat Filter

The top hat filter strips the image of large shapes.

Font Information

acuRead can read strings that use the following fonts:

- SEMI
- Canon
- OCR-A
- Triple
- User-defined fonts

The supported fonts all offer the same character set shown in the table below. User-defined fonts can contain up to 39 characters and must use the ASCII decimal values for the key shown in the table.

Character set	User-defined ASCII decimal codes
uppercase letters (A through Z)	(65 through 90)
digits (0 through 9)	(48 through 57)
dash or minus (-)	(45)
period or decimal point (.)	(46)
space ( )	(32)

OCR/ character set

When searching for each character, acuRead takes into account the following information:

- The size of each character in pixels
- The color of the characters in the string
- The *line error* or maximum perpendicular distance measured in pixels from a character to an ideal baseline. If a character is outside of this maximum distance, then the tool cannot read the character.
- The *space error* or maximum distance measured in pixels that is parallel to the baseline from equal character spacing. If a character is outside of this maximum distance, then the tool cannot read the character.

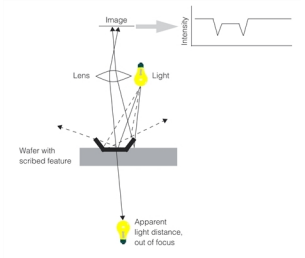
Lighting Control

Many acuRead applications, especially those reading from reflective surfaces, use lighting systems controlled by Cognex hardware. If your application uses such a lighting system, you can control the balance and intensity of the following two kinds of lighting:

- *Brightfield* lighting
- *Darkfield* lighting

Brightfield Lighting

Brightfield lighting places the light on or near the lens axis, as shown in the figure below.

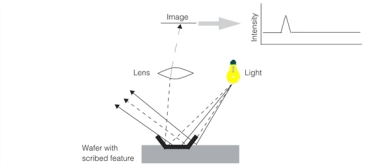


Brightfield lighting

With brightfield lighting, light reflects from the flat wafer surface into the lens, creating a bright background. (The mirror image of the light is out of focus.) Light striking an edge of a scribed character reflects out of the lens, making edges dark. The intensity curve across a stroke is often bimodal.

Darkfield Lighting

Darkfield lighting places the light off the lens axis, as shown in the figure below.



Darkfield lighting

With darkfield lighting, most of the light reflects from the wafer surface away from the lens. However, some of the light striking scribed edges reflects into the lens, which highlights character edges against a dark background. The intensity curve across a stroke often has one peak.

Tuning Lights, Filters, and Character Size

You can specify light, filter, and character size values in the following two ways:

- You can set the values by hand.
- You can direct acuRead to adjust or *tune* any combination of the values.

During tuning, acuRead reads a string, systematically incrementing the settings of the enabled filter, light, and character size options. For each category, acuRead adopts the settings that yield the best scores, reporting the best string when done. You can think of tuning as a kind of multiple-read operation that yields the best possible read settings for filters, lights, and character size. After successful tuning, you typically use the new values for subsequent reads.

You can use tuning to:

- Find good parameter values during development.
- Compensate for run-time changes in the lights, camera, or other production variables.
- Substitute for the read operation to adjust to changing conditions since tuning returns the same string and score results as reading. Note, however, that tuning is much slower than an ordinary read, so you have to balance its potential to improve accuracy with your throughput requirements.

Checksums

acuRead supports the SEMI, BC412, and IBM412 checksums. It also supports the Cognex Virtual Checksum, which reads a string using both Standard and Non-Linear OCR. The checksum passes only if the result strings generated by each method agree.

Virtual Checksum can improve reliability for any wafer scribe that does not have a real checksum and can be helpful with degraded wafer scribes; however, it is less reliable than a Standard OCR checksum.

Note that checksums are not supported for variable length strings.

Character Acceptance Threshold

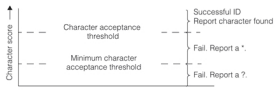
The *character acceptance threshold* is a character score that controls whether an individual character passes or fails. For each field position, acuRead compares the character in the position to each character in the specified font, assigning a score to each comparison. It then compares the highest score with the acceptance threshold and uses the following criteria to determine what to write to the result string:

If the character score is greater than or equal to the character acceptance threshold, acuRead writes the character to the result string.

If the character score is less than the character acceptance threshold but greater than a minimum character acceptance threshold, acuRead writes a "*" to the string. A "*" indicates no match, but suggests that some kind of image is present.

If the character score is less than the minimum character acceptance threshold, acuRead writes a "?" to the string. A "?" indicates no match, and suggests a fundamental problem such as a blank image.

The figure below shows and acceptance threshold and the three possible results.



Acceptance threshold and the result string

The acuRead run-time parameters API allows you to specify an acceptance threshold score for the strings you read. This score applies to the entire string. The character acceptance threshold and minimum character acceptance threshold described in the figure above are derived by acuRead from the string acceptance threshold you specify.

Using acuRead

This section describes how to use acuRead.

Guidelines for Acquiring Images

Before you can read a string, you must perform preliminary steps such as acquiring an image, adjusting the display, and defining the region of interest. The procedures involved depend on whether you are using OMI or CVL. For details about OMI, see the *OMI Object Reference* and the *OMI User's Guide*. For details about CVL, see the *CVL Class Reference* and the *CVL User's Guide*.

You can take the following steps to maximize the quality of the images that your application inspects:

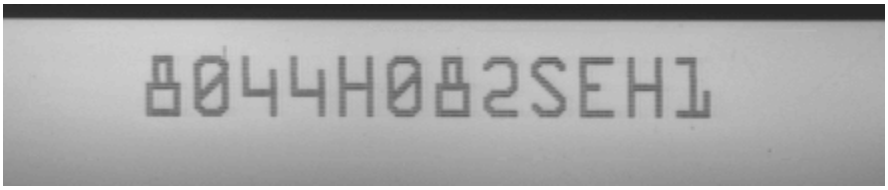
- Place the camera perpendicular to the surface of the part so that the image of the part is not skewed.
- Adjust the camera and lens to obtain a clear, focused, high-contrast image.

  The baseline of the string should be horizontal or rotated by no more than 5 degrees in the image. The character height should be 25 to 40 pixels.
- Ensure that the region of interest contains the entire string.

  A region of interest positioned too tightly at the top and bottom of a string can cause the read to fail or be unreliable, particularly if the characters' horizontal strokes are thin.

  The region of interest should also allow adequate margin at the left and right so that leading and trailing characters are not cut off. Be aware that when using a proportional font, you must allow sufficient ROI space on the right so that if the widest font character were used in the last position, it would fit inside the window.

The figure below shows a representative image.



Clear, focused, horizontal image

Loading a Font

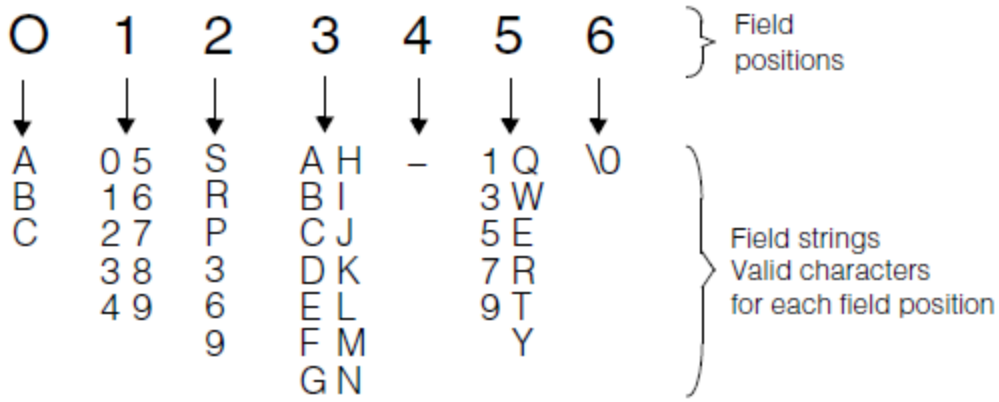
acuRead recognizes characters from a predefined font model. When setting up acuRead, you must specify the font of the characters that the tool will read.

Note: acuRead fonts are in a proprietary format. TrueType, PostScript, and other font formats are not compatible with acuRead.

Defining the String

After selecting a font, you define the number of fields in the string to be read (including checksum fields, if any). This definition is called a *field map* and is composed of a sequence of *field positions* where each field position corresponds to a character position to be read, and each field position contains a *field string* defining the valid characters for that position. (See the figure below).

You can run acuRead in fixed length mode or variable length mode. In fixed length mode the character strings you wish to read are of known length and are all the same length. For these applications you define your field map for the known length and specify the valid field strings for each field position. For example, if you know beforehand that a particular field can contain any digit, but only a digit, then you specify 0 through 9 as its valid characters. Constraining each field to a known set of characters, as shown in the figure below, improves speed and reliability. Specifying the incorrect number of field positions prevents acuRead from finding and reading the string.



Field map example

In the field following the last field position that contains a character, you enter \0 to mark the end of the encoded string. acuRead requires this termination character and ignores any subsequent fields. If you omit the termination character, acuRead assumes that the string is the maximum supported length. Any field strings you do not define are defined by default in the object constructor.

In variable length mode you can read character strings where the number of characters is not always the same. Here you define your field map for the maximum length string you expect to encounter and then specify the field strings for your application. Even though the string length can vary, the field strings that define valid characters for each position are the same for all reads.

Reading Special Characters with OCR

The acuRead tool is able to read special characters with optical character recognition (OCR). This capability:

- Allows you to specify multiple special characters, including '-' (dash), '.' (period), and ' ' (space), in the field string for a field position. acuRead returns a character based on the feature extracted from the image for any of these characters.
- Gives the acuRead tool the ability to distinguish between special characters.

Fonts and Characters Recognized

acuRead can read special characters in the SEMI or OCR-A font.

For a field that includes both alphanumeric and multiple special characters in its field string, acuRead can return any of the following types of characters:

- Alphanumeric characters specified in the field string
- Special characters, including '-' (dash), '.' (period), and ' ' (space), specified in the field string
- '*' (asterisk) or '?' (question mark) characters

For one field position, the following read rules apply:

- For a field with up to one special character in its *field string*, the *result* is the character output by acuRead at this position. The *field string* is the list of characters that are acceptable in this position.
- For a field with multiple special characters in its field string, the result reflects the *true ID* of the character, '?' (question mark), '*' (asterisk), or ' ' (space), at that position, depending on the result score and the field map. The *true ID* is the real character that occupies this position (that is, the test character whose presence you are testing for).

The following table shows the rules used and the result produced by each rule, assuming the use of clean and clear images and correct font initialization:

Read Rule	Example at Specific Position		
	Field String	True ID	Result
1) If the field string contains the true ID, the result is the true ID. The result in this case can also be a '?'(question mark) or an '*' (asterisk) character, depending on the score.	"`.- 0123`"	'`.`'	'`.`'
	"`-. 0123`"	'`-`'	'`-`'
	"`.- 0123`"	'`3`'	'`3`'
	"`.- 0123`"	' '	' '
2) If the field string does not contain the true ID but contains a space, the result is a space.	"`. 0123`"	'`-`'	' '
	"`- 0123`"	'`.`'	' '
	"`. 0123`"	'`4`'	' '
3) If the field string does not contain the true ID but does contain one special character, the result is the special character.	"`-0123`"	' '	'`-`'
	"`-0123`"	'`4`'	'`-`'
4) If none of the above rules apply, the result is a question mark.	"`0123`"	'`4`'	'`?`'
	"`.-0123`"	' '	'`?`'

Operating Ranges

If you are using acuRead with multiple special characters enabled, at least two characters on either end of a string cannot contain special characters. In other words, the field setting of a string should conform to the pattern A^{++} (EA)* A^{++}, where:

- A is a normal field that does not contain special characters.
- E is a field that contains one or more special characters.

- * means zero or more repetitions.
- $^{++}$ means two repetitions.

Because of the intrinsic features of the special characters '.' (period), '-' (dash), and ' ' (space), attempting to read them in noisy images is not recommended. Set field strings to be as restrictive as possible to avoid confusion between special characters and noise.

Selecting a Reading Method

Your choice of Standard OCR or Non-Linear OCR depends on your application. The table below summarizes the criteria that might be used to choose between methods:
(Note that Non-Linear OCR is not supported for user-defined fonts)

Characteristic	Standard OCR	Non-Linear OCR	Comment
Permissiveness	Less	More	Non-Linear OCR often reports a character value where Standard OCR would report a "?" or "*".
Speed	Slightly faster	Slightly slower	
Ability to read severely degraded, partly occluded, or unevenly illuminated characters	Less tolerant	More tolerant of erroneous characters	If string being read has a checksum, use checksum (with either reading method) to increase reliability.

Selection criteria for OCR method

Setting the Acceptance Thresholds

Standard OCR and Non-Linear OCR differ in their scoring algorithms. Non-Linear OCR often yields higher character and string scores than Standard OCR. The default acceptance threshold for Non-Linear OCR is therefore higher than that for Standard OCR, reflecting its higher scores.

The Standard OCR threshold affects the string-search process, the Standard results, and the first phase of Non-Linear OCR because it influences the string and character search process shared by both methods. The Non-Linear threshold affects only the second phase of Non-Linear OCR.

Note: An excessively high Standard OCR threshold can cause Non-Linear OCR to fail if it causes acuRead to fail to find the string in the image.

Each acceptance threshold applies to all of the fields in the string (except for whitespace fields, which are not scored). You generally set the acceptance thresholds to your minimum level of confidence, viewing the string as a whole. The appropriate values depend on image quality, scribe quality, and other considerations. If the string contains a checksum, you can usually set permissive thresholds because the checksum detects misreads.

acuRead uses the Standard OCR and Non-Linear OCR thresholds you specify to derive the character acceptance threshold and minimum character acceptance threshold used to score individual characters. This is discussed in *Character Acceptance Threshold* on page 431.

Enabling a Standard or Virtual Checksum

You should enable the checksum as follows:

- If your string has a SEMI, BC412, or IBM412 checksum, always enable it. If you are using Non-Linear OCR, enabling the standard checksum automatically enables Virtual Checksum as well.
- If your string does not have a standard checksum, specify Virtual Checksum or None.

See *String Scoring* on page 443 for more information about the way that CVL and OMI report checksum results.

Note: Checksums cannot be used in variable length string applications.

Setting the Initial Character Size

For OMI, the valid range for height is 8 through 255 pixels; and for width, 4 through 255 pixels. For CVL, the valid range for height is 8 through 128 pixels and, for width, 8 through 64 pixels. Specifying a value larger than the height or width of the region of interest causes the read to fail.

When searching the preprocessed image for the string, acuRead relies partly on an expected character height and width. Inaccurate values can cause the tool to fail to find the string. When defining the character height and width, choose values that include the outermost pixels in the character strokes, as shown in the figure below.

Setting the character size

Scale Tuning

After finding the first string, acuRead can automatically adjust the character size to the measured size. One use for this adjustment, called *scale tuning*, is to allow the application to dynamically follow slight drift in image size. Enabling scale tuning can increase the robustness of the search algorithm. Disabling scale tuning results in faster operation.

With scale tuning enabled, the specified character height and width values apply only to the first string, and tuning updates the height and width with the measured sizes. With scale tuning disabled, the specified initial values apply to all strings.

Enabling AutoStroke

AutoStroke controls internal step values used when sampling the preprocessed image to find the string. Its benefit depends on character size:

- For characters less than 22 pixels high or 11 pixels wide, enabling AutoStroke improves accuracy, especially if the stroke is thin (less than or equal to about 2 pixels). It also reduces speed by about 20%.
- For characters greater than 29 pixels high or 14 pixels wide, enabling AutoStroke improves speed (up to two times faster, for 23 x 46 characters). However, it can reduce accuracy if the stroke is less than or equal to about 2 pixels.
- Otherwise, AutoStroke has no effect.

You normally enable AutoStroke. Disabling it rarely offers any advantage. Tuning never changes the specified value.

Note: AutoStroke control is available only in OMI. In CVL AutoStroke is always enabled.

Setting the Expected Character Color

When searching the preprocessed image for the string, acuRead relies on an expected character color. The options are:

- Always Black. The characters are always dark on a light background.
- Always White. The characters are always light on a dark background.

- Light Inverse. The character color is the inverse of the current light mode. If the mode is brightfield, the characters are dark. If darkfield, they are light. The Light Inverse option is provided for use with light tuning, so the character color will change when tuning changes the light mode. For more information on brightfield and darkfield lighting, see *Lighting Control* on page 429.

Note: Because light tuning is not appropriate for saved images, Light Inverse should only be used with acquired images.

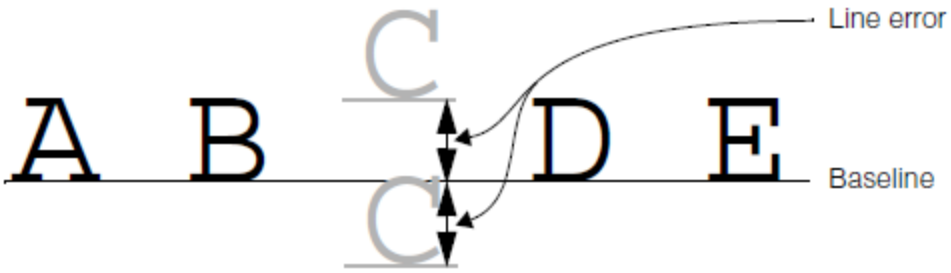
Setting Character Position Tolerances

You can set the number of pixels a candidate character can deviate from its expected position and still be considered a character. This section discusses setting the following tolerance values:

- Line error
- Space error

Setting the Line Error Tolerance

Line error tolerance is the number of pixels that a character can be from an ideal baseline and still be considered a match. As shown in the figure below, acuRead applies the line error tolerance above and below the baseline. The total tolerance is twice the specified line error.



Line Error

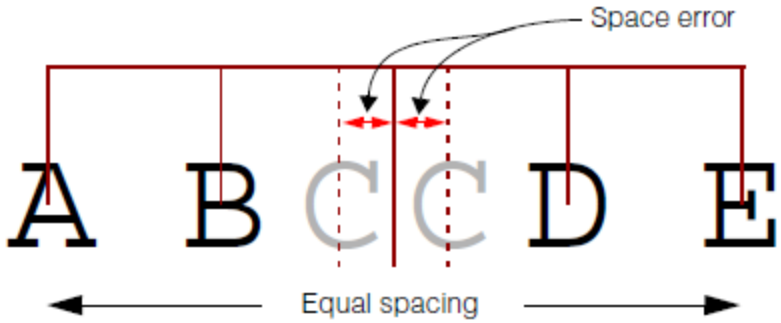
acuRead fits the baseline to the actual image, even if it is slightly skewed, and then applies the line error tolerance, even if not exactly vertical in the image. (In any case, you should try to acquire unskewed images, with baselines no more than 5° from horizontal.)

Line error influences the string search and character search phases of OCR, not the scoring phase. Setting an appropriate value constrains the search and increases the likelihood of obtaining a correct segmentation, in turn reducing the risk of failed reads or misreads.

There usually is not much variation in the vertical position of characters in a wafer scribe. The need to allow for a vertical tolerance reflects the potential for degraded characters. For wafer scribes, this value often can be as small as 2 or 3 pixels. The default value of 2 is good for many applications.

Setting the Space Error Tolerance

Space error tolerance is the number of pixels that a character can deviate left or right from an ideal center line that spaces all of the characters equally. As shown in *Space Error* on page 437, acuRead applies the space error tolerance on both sides of the predicted center line. The total tolerance is twice the specified space error.



Space Error

acuRead fits the center lines to the image even if it is slightly skewed, and then applies the tolerance.

Space error, like line error, influences the string and character search phase of OCR, not the scoring phase. Setting an appropriate value constrains the search and increases the probability of obtaining a correct segmentation, in turn reducing the risk of failed reads or misreads. An excessively large value, especially one above about half the actual character spacing, creates a risk of confusion with adjacent characters.

Wafer scribes are monospaced and generally accurate. The need to allow for a “horizontal” tolerance reflects the potential for degraded characters. For wafer scribes, this value often can be as small as 3 or 4 pixels to handle normal deviation in character spacing and image quality. The default value of 4 is good for many applications.

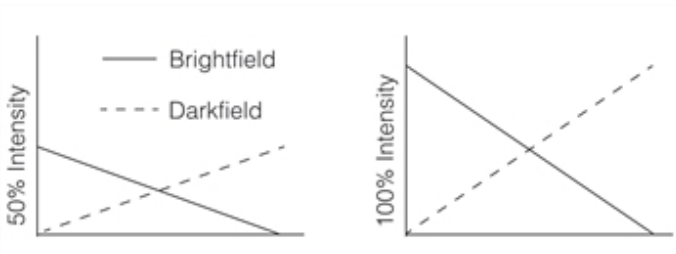
Tuning

This section provides information about tuning the following parameters:

- Brightfield and darkfield lighting
- Filters
- Character height and width (character scale)

Tuning the Lighting

acuRead controls brightfield and darkfield lighting through two parameters, *light intensity* and *light mode*. Light intensity controls the amount of power supplied to the lights. Light mode controls the balance of brightfield and darkfield lighting, which vary in an inverse relationship, as shown in *Darkfield lighting* on page 430.



Light mode and light intensity

Many applications use either brightfield or darkfield lighting, but intermediate values (which add together the light intensity curves) sometimes yield better results.

Finding the best light mode and intensity values can be a complex trial-and-error process. acuRead can greatly simplify the task of setting up the lights through *light tuning*, which automatically tests different values to find the best combination. You can enable tuning for the light mode, light intensity, or both. For each, you specify the range of values to be considered and the step value between trials. Due to refinements in the tuning process, the exact number of trials might differ slightly from the number predicted from the range and step values.

OCR from a reflective surface tends to be sensitive to small changes in lighting, so retuning the light mode, light intensity, or both after a failed read can often improve the result. A common strategy is for the application to try a series of predetermined parameter sets. If all fail, the application retunes the lights, filters, or both. If retuning yields a good read, the new parameter set can be saved and used for subsequent reads. This strategy lets the application dynamically follow changes in the production environment.

Note: Light tuning is intended for use with acquired images, and should not be used with saved images.

Tuning the Filters

acuRead can simplify the process of selecting and configuring filter values through *filter tuning*, which automatically tests different filter values to find the best combination. For each enabled filter, acuRead scores one read with the filter present and another with it absent. The tuned result is the combination of filters that yield the highest string scores.

In most cases, you will want to enable both the clipping and high pass filters. Because high pass and top hat eliminate the same kind of noise, you typically enable only one of them at a time. The high pass filter is faster, but can be less effective than the top hat filter.

Note: OMI refers to filter tuning as “process tuning.” In CVL, the tuning parameter that affects filter tuning is labeled “Ocrf.”

Tuning Character Height and Width

With *scale tuning,* which involves finding the optimal height and width settings for a character, acuRead steps through a range of character heights, widths, or both to find the combination that produces the best string score. You enable tuning for height and width separately.

For information on the supported character heights and widths, see the section *Character Height and Width* on page 443.

Note: Specifying a value larger than the height or width of the region of interest causes the read to fail.

Controlling the Range of Sizes Tested

To control the range of sizes tested, you specify a height range and a width range. The valid values for both ranges are 0 through 16. When you specify a range value, the tool evaluates the values between the current value minus the range value and the current value plus the range value. For example, if the current height is 20 and the range is 2, then tuning evaluates heights 18 through 22.

If the tuning yields an improved value for height or width, acuRead retains the tuned value and uses it as a starting point for the next scale tuning operation. For example, if the initial value for height is 30 and the initial value for range is 5, acuRead uses the range from 25 through 35 for tuning. If the best value after tuning is 32, acuRead adopts 32 as the current height value. The next time acuRead tunes, it starts at 32 and considers 27 through 37. acuRead updates the current value only if tuning improved the score.

During development, you often use scale tuning to refine your initial height and width values, especially since the “best” values sometimes interact slightly with the light and filter settings. In a production application, you can use scale tuning to track slight variations in character size. Tuning the height, width, or both can decrease the risk of failing to find the string; disabling tuning increases processing speed. As always, your accuracy and throughput requirements guide your choice.

Setting Limits for Tuning

Many production OCR applications that benefit from tuning cannot afford to let every tuning operation run to completion. To limit the time required by tuning, you can set the following parameters:

- *ValidLimit*
- *Time2Valid*

- *ScoreLimit*
- *Time2Score*

ValidLimit

ValidLimit is the maximum number (0 to 255) of successful reads that can occur without producing a new best score. “Successful” means that the read produces a string that exceeds the acceptance threshold, conforms to field expectations, and passes the checksum, if any.

When first setting up acuRead, you might prefer a high value, since the goal is to find the ideal tuning and the elapsed time is not critical. In a production application, you might prefer a small value, especially if the string has a checksum. When using a checksum, you can be sure that a successful read does not contain incorrect characters. It often makes sense to stop tuning after the first read that passes, rather than consume production time seeking an ideal solution.

Time2Valid

Time2Valid is the absolute maximum time, in seconds, permitted for tuning. *Time2Valid* is a hard limit: tuning always quits when it reaches this timeout. That is, if *Time2Valid* expires before acuRead reaches *ValidLimit*, then tuning halts after completing the current read. acuRead finishes any read in progress, so the total clock time might slightly exceed the *Time2Valid* value.

ScoreLimit

ScoreLimit is the string score that tuning must reach within *Time2Score* for it to continue. Tuning stops if *Time2Score* elapses without a qualifying read. A *ScoreLimit* is like an acceptance threshold but applied only to tuning and typically set to a lower (more forgiving) value.

OMI and CVL differ in the range of values for *ScoreLimit*. For more information, see the section *ScoreLimit Parameter* on page 444.

Time2Score

Time2Score is the maximum time, in seconds, that can elapse before tuning reaches *ScoreLimit*. Tuning stops if the time elapses without a qualifying read. *Time2Score* is a conditional timeout. *Time2Valid*, in contrast, is unconditional.

ScoreLimit and *Time2Score* define “good enough to keep going.” One use for them is to prevent wasting time trying to tune an image that does not contain a wafer scribe at all. For example, if the *ScoreLimit* is 25 and the *Time2Score* is 10, then the score must exceed 25 within 10 seconds or tuning stops.

Initiating a Read

After you have defined and tested the acuRead parameters, you are ready to read a string. First, you acquire a new image and then trigger a read. acuRead preprocesses the region of interest with the enabled filters, finds the string, compares each field to the loaded font, computes character scores, selects the best characters, and computes the string, string score, and other results.

You can also use the tuning operation as an alternative to the read operation.

Getting the Results

After a read or tune operation, acuRead returns the following results:

- The string, including both data and checksum fields. acuRead marks failed fields with a “*” (below the user-defined acceptance threshold, but above its minimum) or a “?” (below the minimum threshold).
- The length, in fields, of the string. Includes both data and checksum fields.

- The string score and checksum results:
  - CVL reports distinct string score and checksum results.
  - OMI combines the string score and checksum (described in the section *String Scoring* on page 443).
- The value, location, and character score of the individual characters in the string:
  - In OMI, this data is reported to an array of ORC (for “optically recognized characters”) structures.
  - In CVL, the equivalent data is reported through the **ccAcuReadResult** class.

Additional results for tuning include:

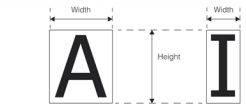
- The best string score obtained during tuning
- The character string with the best string score obtained during tuning

User-defined Font Specifications

When you define a font set for use with acuRead, the fonts must conform to the following specifications. These specifications are designed for machine readability and not for human readability or visual appeal. All measurements are in pixels.

Font Width and Height

Each character in the font set is contained within a bounding box. The bounding box height for all characters must be the same while the bounding box width can vary depending on the character. See *Font height and width* on page 440.

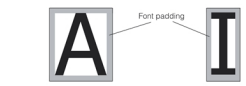


Font height and width

The font size is the width of the widest character bounding box.

Font Padding

The characters you define within the bounding box can be any size and may or may not touch the bounding box itself. Most fonts leave a space around the character called padding. Each character can have unique padding but it is common to use the same padding for consistency of appearance. See *Font padding* on page 440.



Font padding

The following are some general guidelines for padding.

- In general use less padding rather than more. Larger padding results in a slower read operation since there are more pixels to process.
- Although characters before padding may have different height, you should pad the top and/or bottom so that the bounding boxes of all padded characters have the same height. Also, adjust the padding so that when all the character bounding boxes are aligned vertically, all of the characters have a common baseline.
- The side padding for the width dimension depends on the shape of the characters. A good rule is to use the side padding to balance the amount of background and foreground information. Keep in mind that too much padding will degrade the read performance. You may wish to determine the side padding value empirically.

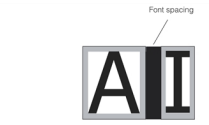
- If you are converting a proportional width font to fixed width font by padding along the width dimension, keep in mind that too much padding will degrade the read performance. Also, keep in mind that acuRead may fail if the inter-character spacing is too large.

These guidelines work best in applications where you are designing the printed fonts as well as designing the OCR system. Here more padding simply spaces out the characters and increases the number of pixels that must be processed.

However, be aware that in applications where you are reading character strings printed by equipment you do not control, you need to tailor the padding to the fonts you will read. Too much padding can cause acuRead to fail rather than just increasing compute time. Choose padding so that the inter-character spacing remains roughly constant throughout the string, while providing each character with some side padding, at least 1-3 pixels on each side with more padding for larger characters.

Font Spacing

Font spacing is the distance between the facing edges of the bounding boxes of two adjacent characters. This value is implicit. In a character string produced from a font, the between-character spacing is expected to remain uniform over the entire string. Any deviation is considered space error. The tool will tolerate very small space errors. See *Font spacing* on page 441.



Font spacing

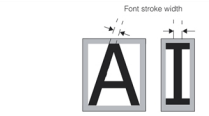
Word Spacing

The minimum word spacing is the width of a space character bounding box plus the width of two font spaces, one preceding the space character and one following it. The width of the space character bounding box should not be less than the narrowest bounding box for any other character.

There is no maximum word spacing. Note however, that when the word spacing is larger than the minimum, the tool reads the gap as multiple spaces, and inserts multiple space characters into the returned text string.

Font Stroke Width

The width of character strokes is an important element of a font. The acuRead tool requires uniform stroke width of at least one pixel. The font characters shown in *Font stroke width* on page 441 satisfy this requirement.



Font stroke width

The figure below is an example of a font that does not satisfy the stroke width requirement for consistency. Some parts of the stroke width are much smaller that others.

Example of inconsistent stroke width

Character Confusion

The confusion score between two characters indicates the probability that the characters can be confused with one another during a read operation. The confusion score is a number between 0 and 1 where 0 means that the characters will never be confused with each other and 1 means that they will always be confused with each other.

Confusion scores for all character pairs in your font set can be obtained from the **ccOCAlphabet** object you use to access a user-defined font (**ccAcuReadFont::access()**). When integrating a new user-defined font into your application you should check all the possible confusion character combinations you may encounter. If you have a serious confusion problem you may wish to choose a different font.

Note that this is application dependent. For example, you may have a high probability of confusion between an I and a 1 but if you application is known to never read numeric characters you can train it for I only.

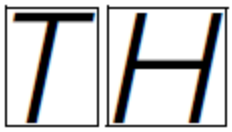
User-defined Font Operating Ranges

The following operating ranges apply to user-defined fonts.

Training

- Font type:

  Italic or slanted fonts can be used if the image aligned bounding box of neighboring characters in a string do not overlap. For example, see *Valid use of italic font* on page 442.

  Valid use of italic font

  acuRead may not be able to read strings printed with kerning.

- Proportional ratio R_p
  The ratio of the bounding box width of the widest character to the narrowest character.

  $1 <= R_p <= 5$

  For fixed width fonts, $R_p = 1$.

- Font size:

  Font height >= 10 pixels. All characters in a given font must be the same height.

  Font width >= 7 pixels.

  Font area <= 1024 pixels.

  Font stroke width >= 1 pixel and consistent throughout the font set.

Run Time

The following ranges are related to how the string appears in the acquired image.

- Character size:

  8 <= width <= 64 pixels; 8 <= height <= 128 pixels.

- Inter-character spacing (s):

  s >= 1 pixel. Inter-character spacing should be less than the width of the space character.

- Space character:

  The space character width must be equal to, or greater than, the width of the most narrow font character.

- Word spacing:

  The minimum word spacing is the width of the space character plus two inter-character spaces.

- Character size scale:

  Run-time characters cannot be smaller than trained characters.

  Run-time characters can be larger than trained characters but cannot exceed the run-time character size specified in the first bullet above. Note that run-time characters can have an aspect ratio different from the trained character as long as they are within the minimum and maximum size limits.

- String rotation (r):

  r <= 5 degrees.

- String length (l):

  3 <= l <= 30 characters.

- Run-time image ROI size:

  64 <= width <= 2048 pixels. 64 <= height <= 2048 pixels.

Variable Length Strings

The following guidelines are for variable length string applications.

- The acuRead variable length string mode works best for high signal-to-noise ratio applications.
- When you run in variable length string mode, your application will be more robust if you make your field strings more general rather than restricting field strings to a smaller set of characters.

Framework Differences

This section describes differences in the availability or interface of acuRead features in the OMI and CVL frameworks.

Character Height and Width

CVL supports character heights from 8 through 128 pixels; widths, from 8 through 64 pixels. OMI supports character heights from 8 through 255 pixels; widths, from 4 through 255 pixels.

AutoStroke

AutoStroke control is available only in OMI. In CVL AutoStroke is always enabled.

String Scoring

CVL reports the string score as a number between 0.0 and 1.0. Checksum results (if any) are reported separately from the string score.

When Standard OCR is the selected recognition method in OMI, acuRead adds 100 to the string score when a SEMI, BC412, or IBM412 checksum passes.

When Non-Linear OCR is the selected recognition method in OMI, acuRead combines the string score and the checksum result as follows:

- No checksum: add nothing to the Non-Linear OCR score and report the Non-Linear result string.
- Virtual Checksum: compare the Non-Linear OCR and Standard OCR result strings. If they agree, add 100 to the Non-Linear string score. Report the Non-Linear string.
- SEMI, BC412, or IBM412 checksum:

Std Passes	NL Passes	Returns
Yes	Yes	NL score + 200, report NL string
No	Yes	NL score + 100, report NL string
Yes	No	Std score + 100, report Std string
No	No	NL score + 0, report NL string

Tuning

This section discusses differences in the tuning operation between the OMI and CVL versions of acuRead.

Tuning All Parameters

The OMI version of acuRead supports a combined tuning operation (“tune all”) based on the currently enabled light-, filter-, and scale-tuning options. An exhaustive search might take too much time to be practical, so “tune all” intelligently abbreviates the search. Like the individual tuning operations, “tune all” finds the best string score and preserves the parameter values that yielded it.

There is no “tune all” option in CVL; however, the CVL interface for tuning consolidates the tuning of lights, filters, and character scaling in a single command such that the default behavior in CVL is equivalent to “tune all.”

ScoreLimit Parameter

The values of the *ScoreLimit* parameter that specifies the string score that tuning must reach within *Time2Score* differ in CVL and OMI.

In the CVL version of acuRead, the supported range is from 0.0 through 1.0. For the OMI version of acuRead, the supported range is from 0 through 100. Any extra points that might be added to a string score when a checksum passes are ignored here.

Light Intensity and Light Mode

OMI and CVL parameters for setting light intensity differ in scale. In OMI, light intensity can range from 0 (no power) through 255 (full power). In CVL, light intensity can range from 0 (no power) to 1.0 (full power). OMI and CVL both refer to their light intensity parameters as “light power.”

OMI and CVL parameters for setting the light mode differ in both scale and sense. In OMI, the light mode value scales from 0 to 255, where 0 is brightfield-only and 255 is darkfield-only. In CVL, the equivalent “bright field power ratio” parameter scales from 0.0 to 1.0, where 0.0 is darkfield-only and 1.0 is brightfield-only.

Symbol Tool

The Symbol tool is a vision tool that reads 2D Data Matrix symbols and QR Code symbols. It first builds a model of a typical symbol. Then, using the model and several values that control how to apply it, the Symbol tool locates and decodes symbols.

This chapter provides the background for using the Symbol tool.

Some Useful Definitions on page 445 is a glossary of relevant terms.

Overview of the Symbol Tool on page 446 outlines how the Symbol tool works.

Symbol Tool Operating Modes on page 446 describes the operating modes supported by the Symbol tool and how your system's security determines the tool's behavior.

Symbologies Supported by the Symbol Tool on page 447 describes the basic features of 2D Data Matrix symbols and QR Code symbols.

Using the Symbol Tool on page 451 describes how to configure the Symbol tool for use in an application.

Developing an Application on page 460 provides information about developing an application using the Symbol tool.

Framework Differences on page 462 summarizes the differences between the OMI and CVL versions of the Symbol tool.

Some Useful Definitions

acceptance threshold: A score used by the Symbol tool to determine whether a run-time image is a symbol or not.

AIM: The standards organization that publishes barcode and 2D symbol specifications, including the Data Matrix and QR Code specifications.

codeword: A symbol character value.

confusion threshold: A score that defines success when using the model to search for the finder pattern. A match scoring above this threshold automatically succeeds.

Data Matrix: A 2D matrix symbology made up of modules arranged in a square or rectangle within a perimeter finder pattern.

ECC: Error Checking and Correction, a mathematical check for correct decoding.

encoding scheme: A set of characters that can be encoded in a symbol.

finder pattern: A unique marking used to find, scale, and orient an image of a symbol. The Data Matrix finder pattern is an L-shaped, one-cell perimeter around the data area, dark on two adjacent sides, and alternating light and dark on the opposite sides. The QR Code finder pattern is a set of three square position-detection patterns.

learning: The Symbol tool operation that detects symbol features, builds a symbol model, and then optimizes that model.

model: A generalized example of the symbol to be found in the image.

module: A single cell in a symbol. It is nominally a square or a round dot.

points: The coordinate locations of the three targets that define the symbol's finder pattern.

QR Code: A 2D matrix symbology made up of modules arranged in a square with a unique finder pattern located at three corners of the square.

quiet zone: The blank perimeter around a symbol, to isolate it from its background.

symbol: A 2D matrix used to encode data. Includes the quiet zone, finder pattern, and data area.

symbology: A set of rules that define a format for encoding and decoding 2D symbols.

Overview of the Symbol Tool

The Symbol tool locates 2D Data Matrix or QR Code symbols in an image and returns the string encoded by the symbol. Using the Symbol tool, you can develop applications that decode symbols that are applied to reflective or non-reflective surfaces.

A typical application has a *learning phase* and a *decoding phase*. In the learning phase, the tool locates a symbol, learns the values of specified parameters, builds a model from this data and data that you provide, and decodes the symbol. This model is subsequently used to locate and decode other symbols.

In the decoding phase, the tool decodes the symbol.

To write an application, you perform the following steps:

1. You write code to acquire images of symbols.
2. You create a Symbol tool with an appropriate operating mode and specify the kind of symbols that the tool will decode: 2D Data Matrix or QR Code symbols.
3. You specify which of several *Learn parameters* that the tool will learn. Examples of Learn parameters include ECC level, grid size, and polarity.

   You also specify values for those Learn parameters that the tool will not learn.
4. You specify the values of *Find parameters*, depending on the chosen operating mode, which the tool uses to locate and orient each run-time symbol. Examples include angle range, scale range, confusion threshold, and acceptance threshold.
5. You call a learn function to locate a symbol in a typical image, learn required parameters, and create a model of the symbol. The learn function also extracts the string encoded in the 2D symbol. Calling the learn function initiates the learning phase of the application.
6. (Optional) If you are using a lighting module to read symbols from semiconductor wafers or other reflective surfaces, you call the tune function to determine optimum lighting settings.
7. You call the decode function to search each run-time image for a symbol and then decode it. This function uses parameters learned during the learning phase and parameters that are set directly. This begins the decoding phase of the application.

In the decoding phase, the tool locates an area in an image that may contain a symbol and computes the correlation between the image area containing the candidate symbol and the model. Depending on the score, the tool performs one of the following actions:

- If the score is below the acceptance threshold, it rejects the symbol and continues searching for other possible symbols.
- If the score is between the acceptance and confusion thresholds, the tool examines the symbol further to determine if it is a symbol.

  If the symbol is valid, the tool decodes the symbol and returns the decoded string and data about the symbol.
- If the score exceeds the confusion threshold, the tool decodes the symbol and returns the string and data about the symbol.

Symbol Tool Operating Modes

The Symbol tool supports three operating modes:

- Standard

  Use standard operating mode for wafer and degraded direct-mark applications where there is the possibility of low contrast or degradation within the symbol region and low clutter or confusion outside the symbol region.

- High-contrast

  Use high-contrast operating mode for label, PCB, and high-contrast direct-mark applications where there is high-contrast within the symbol region and possibly high clutter or confusion outside the symbol region.

- Autodetect

  Use autodetect mode to enable the tool to select the appropriate operating mode based on your system security.

The Symbol tool's operating mode depends on the level of hardware security present in your system. Two security levels are available:

- Standard security uses the *2DSymbol* security bit
- High-contrast security uses the *2DSymbolHighContrast* security bit

To operate the Symbol tool in standard mode, you require standard security enabled for your system. To operate in high-contrast mode, you require high-contrast security enabled for your system. *Operating modes, system security, and Symbol tool behavior* on page 447 summarizes the interaction of operating modes, hardware security, and tool behavior.

Operating mode	Security	Tool behavior
Standard	Standard	Operates in standard mode
	High-contrast	Throws an exception
	Standard and high-contrast	Operates in standard mode
High-contrast	Standard	Throws an exception
	High-contrast	Operates in high-contrast mode
	Standard and high-contrast	Operates in high-contrast mode
Autodetect	Standard	Operates in standard mode
	High-contrast	Operates in high-contrast mode
	Standard and high-contrast	Operates in combined mode

Operating modes, system security, and Symbol tool behavior

Symbologies Supported by the Symbol Tool

The Symbol tool decodes the following types of symbols:

- Data Matrix
- QR Code

Data Matrix

Data Matrix is a 2D symbology that features a rectangular or square array of dark and white modules used to encode data. The symbol has a finder pattern, which is used to locate and orient the pattern. This pattern consists of a vertical and horizontal bar that is one module wide on the left and bottom sides of the symbol and alternate dark and light modules on the right and top sides. On the outer four sides of the symbol is a quiet zone that is one module wide for continuous Data Matrix symbols and four modules wide for Data Matrix symbols with dot-matrix format.

Data Matrix symbol on page 447 illustrates a Data Matrix symbol:

Data Matrix symbol

There are two primary groups of Data Matrix symbols, which are classified according to their error checking and correction schemes.

- ECC-000 through 140, which are square and have an odd number of data marks on each side
- ECC-200, which have an even number of data marks on each side and can be square or rectangular.

Note: The 2D Symbol tool supports Data Matrix macro characters.

QR Code

QR Code (Quick Response Code) is a two-dimensional matrix symbology having finder patterns on three of its corners.

There are three versions of QR Code symbols:

- Model 1: original version
- Model 2: enhanced version
- Micro QR

QR Code symbols can represent the following types of data:

- Numeric (digits 0 - 9)
- Alphanumeric
- 8-bit
- Kanji

QR Code symbol on page 448 illustrates a QR Code symbol:

QR Code symbol

For a detailed description of QR Code symbols refer to the following industrial specifications:

- AIM International Symbology Specification 97-001
- ISO/IEC 18004:2000 Information technology - Automatic identification and data capture techniques - Bar code symbology - QR Code
- ISO/IEC 18004:2005 Information technology - Automatic identification and data capture techniques - QR Code 2005 barcode symbology specification

The first two specifications apply to Model 1 and Model 2 QR Code symbols. The third specification applies to Model 2 and Micro QR Code symbols. The following summary is taken from the ISO/IEC 18004:2005 specification.

“It is necessary to distinguish four technically different, but closely related members of the QR Code family, which represent an evolutionary sequence.

- QR Code Model 1 was the original specification for QR Code and is described in AIM International Symbology Specification 97-001.

- QR Code Model 2 was an enhanced form of the symbology with additional features (primarily the addition of alignment patterns to assist navigation in larger symbols), and was the basis of the first edition of this International Standard.

- QR Code 2005 (the basis of this edition of this International Standard) is closely similar to QR Code Model 2 and, in its QR Code format, differs only in the addition of the facility for symbols to appear in a mirror image orientation, for reflectance reversal (light symbols on dark backgrounds) and the option for specifying alternative character sets to the default.

- The Micro QR Code format (also specified in this International Standard), is a variant of QR Code 2005 with a reduced number of overhead modules and a restricted range of sizes, which enables small to moderate amounts of data to be

represented in a small symbol, particularly suited to direct marking on parts and components and to applications where the space available for the symbol is severely restricted.

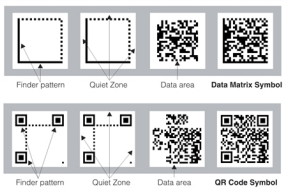
QR Code 2005 is a matrix symbology. The symbols consist of an array of nominally square modules arranged in an overall square pattern, including a unique finder pattern located at three corners of the symbol (in Micro QR Code symbols, at a single corner) and intended to assist in easy location of its position, size and inclination. A wide range of sizes of symbol is provided for together with four levels of error correction. Module dimensions are user-specified to enable symbol production by a wide variety of techniques.

QR Code Model 2 symbols are fully compatible with QR Code 2005 reading systems.

Model 1 QR Code symbols are recommended only to be used in closed system applications and it is not a requirement that equipment complying with this standard should support Model 1. Since QR Code 2005 is the recommended model for new, open systems application of QR Code, this International Standard describes QR Code 2005 fully, and lists the features in which Model 1 QR Code differs from QR Code 2005 in Annex N."

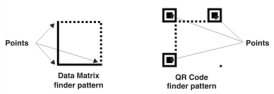
Common Features of 2D Symbologies

Both Data Matrix and QR Code symbols consist of a *finder pattern*, a *data area, and a quiet zone*. *Data Matrix and QR Code symbols* on page 449 contrasts a Data Matrix symbol with a QR Code symbol:



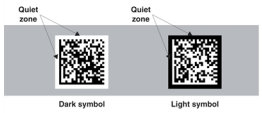
Data Matrix and QR Code symbols

The finder pattern is used to locate, scale, and orient the symbol. As illustrated in *Point locations in Data Matrix and QR Code symbols* on page 449, three points uniquely define the finder pattern.



Point locations in Data Matrix and QR Code symbols

A 2D symbol has one of two *polarities*: light-on-dark or dark-on-light, as illustrated in *Dark vs. Light Data Matrix symbols* on page 449.



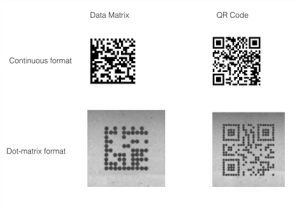
Dark vs. Light Data Matrix symbols

The symbol color is the inverse of the background color. White paper, for example, requires a dark symbol.

In symbols with dark-on-light polarity, data is encoded as follows:

- A dark module represents a binary one.
- A light module represents a binary zero.

Data Matrix and QR Code symbols can be produced in either continuous or dot-matrix formats. *Continuous and dot-matrix formats* on page 450 illustrates Data Matrix and QR Code symbols produced in both formats:



Continuous and dot-matrix formats

The *quiet zone* for a continuous format symbol is one module wide for Data Matrix symbols, four modules wide for QR Code, Model 1 and Model 2 symbols, and two modules wide for Micro QR symbols. The quiet zone for symbols produced using the dot-matrix format is four modules wide. For more information about quite zones see the ISO specifications described *AIM International Symbology Specification 97-001* on page 448.

Unique Features of Data Matrix Symbols

This section summarizes the unique features of Data Matrix symbols.

AIM Compliance

Most Data Matrix symbols comply with AIM standard, but some use older encoding schemes that precede the standard. The Symbol tool can decode AIM-compliant symbols and some noncompliant symbols.

Rectangular or Square Shape

A Data Matrix symbol can be square or rectangular, as shown in *Square and rectangular Data Matrix symbols* on page 450.

Square and rectangular Data Matrix symbols

Grid Size

The grid sizes for Data Matrix symbols can differ even when the symbol encodes the same information. *Data Matrix symbols in three sizes* on page 450 illustrates three symbols that encode the same string but differ in grid size.

Data Matrix symbols in three sizes

The Symbol tool supports the following grid size ranges for Data Matrix symbols:

- Square Data Matrix: from 9x9 modules to 144x144 modules.
- Rectangular Data Matrix: the legal combinations are 8x18, 8x32, 12x26, 12x36, 16x36, and 16x48.

The row and column values include the finder pattern but exclude the quiet zone.

Error Checking and Correction (ECC) methods

The Symbol tool supports two Error Checking and Correction (ECC) methods: convolutional coding (for ECC 50, 80, 100, and 140) and Reed-Solomon encoding (for ECC 200). A higher ECC level allows data recovery despite an increasing amount of damaged or unreadable symbol area, but also reduces the amount of data that a symbol can represent.

Unique Features of QR Code Symbols

This section summarizes the unique features of QR Code symbols.

AIM Compliance

The Symbol tool decodes all AIM-compliant QR Code whose symbol size is less than or equal to 49 modules per side.

Grid Size

The Symbol tool supports the following grid size ranges for QR Code symbols: from 21x21 modules to 49x49 modules in increments of 4. The legal values are 21, 25, 29, 33, 37, 41, 45, and 49. The grid size values include the finder pattern but exclude the quiet zone.

Reed-Solomon Encoding

The QR Code symbology supports four levels of Reed-Solomon encoding. It also explicitly supports level detection, eliminating the need to set the ECC level for decoding.

Model Layout

QR Code symbology has three model layouts: Model 1 (the original version), Model 2 (the enhanced version) and Micro QR. See *Various model layouts of QR Code* on page 451.

Various model layouts of QR Code

Using the Symbol Tool

This section discusses the parameters that are used with the Symbol tool.

Specifying the Operating Mode

To optimize the performance of the Symbol tool, you should select an appropriate operating mode. The selection of operating modes depends on available hardware security support. Use the guidelines listed in the table below to select an operating mode.

Operating Mode	Use If
Standard	• The tool is to be used on wafer or degraded direct-marks. • The application requires calling the learn function once on an image and then using the decode function to decode all run-time symbols whose scales are significantly different from the learned symbol.
High-contrast	• The tool is to be used on paper labels or other high-contrast direct-marks where symbols have clearly defined features and relatively clean local background. • The application requires calling the learn function to decode every symbol because conditions may change at any time. Using either high-contrast or autodetect mode may decrease the tool's average processing time. • The application requires learning the nominal grid from a dot-matrix symbol.
Autodetect	• The symbol images are of unknown types. • The application requires calling the learn function to decode every symbol because conditions may change at any time. Using either autodetect or high-contrast mode may decrease the tool's average processing time.

Guidelines for selecting a Symbol tool operating mode

Specifying the Symbology

To build a correct model, you must specify the kind of symbol to decode, either Data Matrix or QR Code. The tool supports the following symbologies:

- AIM-compliant Data Matrix
- Non-AIM Data Matrix

  The tool can decode AIM 000, 050, 080, 100, 140, or 200 symbols that have an incorrect number of rows and columns. The tool cannot decode symbols that differ from the AIM standard in other aspects.
- AIM-compliant QR Code

You must always assign the symbology for your application because the tool cannot learn it. If you specify the incorrect symbology, the Symbol tool builds an incorrect model and cannot find or decode the symbol.

Specifying Learn Parameters

This section describes the set of parameters that the Symbol tool uses to create a model of a symbol. The tool then uses the model to locate and evaluate each run-time symbol. You must specify a few learn parameters such as an AIM compliance flag. You can also specify that the tool learn some or all of the parameters that do not need to be set.

Data Matrix Learn Parameters

For Data Matrix symbols, the tool can learn the following parameters:

- The Error Checking and Correction (ECC) type
- The grid size, which is height and width in modules
- The nominal grid, which is the affine rectangle which defines the 2D symbol grid
- The scale of the symbol relative to the model (The tool learns this parameter if you specify that it learn the nominal grid.)
- The angle of the symbol relative to the model (The tool learns this parameter if you specify that it learn the nominal grid.)
- The polarity of the symbol, which is either light-on-dark or dark-on-light
- The optimization of the symbol model's pixel height and width and the position of the finder pattern

If you have not selected the high-contrast operating mode, you must specify the following parameters:

- AIM compliance flag, which indicates whether the Data Matrix symbol is AIM compliant or not
- Mirror flag, which indicates whether the image of the symbol is mirrored or not

QR Code Learn Parameters

For QR Code symbols, the parameters that the tool can learn are:

- The grid size, which is height and width in modules
- The nominal grid, which is the affine rectangle which defines the 2D symbol grid
- The scale of the symbol relative to the model (The tool learns this parameter if you specify that it learn the nominal grid.)
- The angle of the symbol relative to the model (The tool learns this parameter if you specify that it learn the nominal grid.)
- The polarity of the symbol, which is either light-on-dark or dark-on-light

- The optimization of the symbol model's pixel height and width and the position of the finder pattern
- The model type to look for when learning

If you have not selected the high-contrast operating mode, you must specify the following parameters:

- Mirror flag, which indicates whether the image of the symbol is mirrored or not

Setting or Learning ECC Level (Data Matrix only)

This section discusses the criteria that you can use to decide between setting the ECC level for Data Matrix symbols or having the tool learn the ECC level. The QR Code symbology supports four levels of Reed-Solomon encoding. It also explicitly supports level detection, eliminating the need to set the ECC level for decoding.

The Data Matrix symbology supports two Error Checking and Correction (ECC) methods, convolutional coding for ECC 000, 050, 080, 100, and 140 and Reed-Solomon encoding for ECC 200. A higher ECC level allows data recovery despite an increasing amount of damaged or unreadable symbol area, but it also reduces the number of codewords that a symbol of a given size can hold.

The ECC level is part of the symbol model. You can specify the expected ECC level or have the tool learn it. If you know the ECC level beforehand, setting it eliminates the processing required to detect it. Otherwise, you should have the tool learn the ECC level.

Setting or Learning the Model Layout (QR Code)

You should configure the tool to learn the model layout for QR Code symbology, either Model 1 (the original version) or Model 2 (the enhanced version).

Setting or Learning the Grid Size

Data Matrix and QR Code symbologies support symbols that differ in grid size, which is the height and width in modules, as distinct from the image size in pixels.

Grid size ranges on page 453 summarizes the supported grid sizes:

Symbol Type	Size Range
Square Data Matrix	9x9 modules to 144x144
Rectangular Data Matrix	8 to 16 rows 18 to 144 columns Legal combinations are 8x18, 8x32, 12x26, 12x36, 16x36, and 16x48
QR Code	For Model 1 and Model 2: • 21x21 to 49x49 in increments of 4 • Legal values are 21, 25, 29, 33, 37, 41, 45, and 49. For Micro QR: • 11x11 through 17x17 in increments of 2

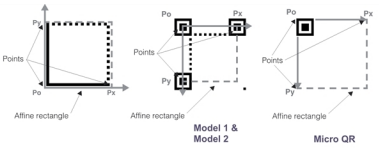
Grid size ranges

Grid size is part of the symbol model. If you know the grid size of the symbols that your application will decode, then set the grid size by specifying the number of rows and columns in the grid. If you do not know the size, then have the Symbol tool learn the grid size.

Setting or Learning the Nominal Grid

The nominal grid is an affine rectangle that defines the dimensions and orientation of the finder pattern for both Data Matrix and QR Code symbols. The Symbol tool relies on the point locations provided by the nominal grid to precisely locate the finder pattern in the image. Given an image of reasonable quality, the Symbol tool can automatically detect the point locations during the learning phase.

The top two and left-bottom corner points of the affine rectangle define the location of the finder pattern, as shown in the figure below.



Affine rectangles superimposed on Data Matrix and QR Code finder patterns

The specification of an affine rectangle allows variations in the scale, orientation and aspect ratio of the scanned symbol. Configuring the Symbol tool to learn the nominal grid is often more accurate than specifying the nominal grid.

Note: The handedness and orientation of the affine rectangle reflect the handedness and orientation of the symbol. For QR Code symbols, the affine rectangle represents a rotated left-handed coordinate system while for Data matrix symbols, the affine rectangle represents a right-handed coordinate system.

If you do not configure the tool to learn the nominal grid, you can either:

- Set the nominal grid in the application
- Create a user interface for an operator to specify the nominal grid

Setting or Learning Angle and Scale

The angle parameter specifies the expected angle of the symbol being located relative to the angle of the model. The scale parameter specifies the ratio between the expected scale of the symbol being located and the scale of the model. The scale parameter is relative to the client coordinate space of the image.

If you configure the tool to learn the nominal grid, then it will learn the angle parameter. Otherwise, if you know the angle value, set it explicitly.

Setting or Learning Polarity

A 2D symbol can be either light-on-dark or dark-on-light. The symbol color is the inverse of the background color. As part of the symbol model, you can assign polarity or have the tool learn it. For symbols printed on matte surfaces such as paper, you often know the expected polarity and can assign it to eliminate the processing required to learn it. For symbols marked or scribed on metallic or reflective surfaces such as silicon wafers, the polarity often depends on the lighting and might change if the lighting changes. In this case, it is better for the Symbol tool to learn the polarity.

Setting or Learning Model Optimization

You can enable or disable optimization of the symbol model's pixel height and width, as well as the position of the finder pattern. You should generally enable optimization. On exceptionally well-defined symbols, disabling it may decrease learning time.

Mirroring the Image

In a production system, you sometimes reflect an image with a mirror; or with transparent surfaces, you sometimes read the image from the back. Both situations reverse the image as shown in *A 2D symbol and its mirror image* on page 454.

A 2D symbol and its mirror image

The Symbol tool assumes that the symbol image is normally oriented and requires more processing time to build a model from a mirrored image. If your images are mirrored and you have chosen not to use the high-contrast operating mode, you must specify that the tool flip the image before processing it.

Tuning Light Parameters (Optional)

If you are using the Symbol tool to read and decode symbols that are scribed or etched on reflective surfaces such as semiconductor wafers, you can use the Symbol tool's tune function to determine optimum values for a lighting module such as Cognex's acuLight II or ultraLight II.

Most lighting modules allow you to vary the illumination mode and illumination intensity. The available illumination modes are *darkfield* lighting (which uses off-axis lighting and forms images of light features on a dark background) and *brightfield* lighting (which uses on-axis lighting and forms images of dark features on a light background).

During tuning, the Symbol tool repeatedly reads and decodes a symbol while varying the illumination mode and intensity (using the ranges and increments that you specify). The tune function returns the combination of settings that produced the best decode result.

Specifying Find Parameters

Find parameters influence how the Symbol tool searches for and applies the model to the image. When not operating the tool in high-contrast mode, you set the following Find parameters:

- Acceptance threshold
- Confusion threshold
- Contrast threshold
- Aspect ratio
- Angle and angle range
- Scale and scale range
- Perspective distortion enabled
- Non-conformant modules enabled

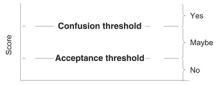
In high-contrast operating mode, the Symbol tool ignores all Find parameters, and instead uses an angle range of 180 degrees and a scale range of 5 percent.

Setting the Acceptance and Confusion Thresholds

The acceptance and confusion thresholds are score values that control whether an image area matches or fails to match the model. The *acceptance threshold* defines the lowest score for a valid match. Image areas with scores below the acceptance value are not matches, so the Symbol tool rejects them as potential symbols.

The *confusion threshold* defines the highest score for an invalid match. Areas with scores above the confusion value are matches, and the Symbol tool immediately accepts them as potential symbol locations.

As shown in *Acceptance and confusions thresholds, 2D symbols* on page 455, scores between the two thresholds might or might not match the model.



Acceptance and confusions thresholds, 2D symbols

The valid range for both thresholds differs in CVL and OMI. For CVL it is from 0.0 to 1.0. For OMI it is 20 to 99. The defaults are appropriate for most applications and rarely require adjustment. If you do change the values, always set the confusion value above the acceptance value. For information about the defaults for these parameters, see the section *Framework Differences* on page 462.

Setting the Contrast Threshold

The *contrast threshold* is the minimum contrast that a run-time image may have and still be considered a symbol. This threshold is a fraction of the contrast of the symbol model.

Using the contrast threshold, your application can reject overly “dim” images such as badly degraded images of symbols or smudges that might be mistaken for symbols. *Contrast Threshold, 2D symbols* on page 456 shows images that pass or fail depending on the contrast threshold.

Contrast Threshold, 2D symbols

Setting the Aspect Ratio

The *aspect ratio* is the expected ratio of pixel width to pixel height. You should use the default value except with unusual cameras or single-field acquires.

Setting Angle Reference

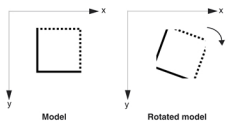
If your run-time images are rotated by a constant factor, you can have the Symbol tool return the angle less the amount of a reference angle that you specify. This reference angle is a vector whose x and y components are specified with respect to the x-axis of the client coordinate system. All angles are measured with respect to the specified reference orientation.

For example, an application inspects a part whose 2D symbol is always tilted by 3 degrees. In addition, the camera is rotated by an angle of 90 degrees so that the actual angle of the symbol is 93 degrees. However, since the camera rotation is constant, the application specifies a reference angle of 90 degrees, which directs the Symbol tool to return an angle for each run-time symbol of 3 degrees, (93 – the reference angle).

To specify the reference angle, you specify the x and y components of a vector that defines the angle with respect to the x-axis of the client coordinate system. For example, (1, 1) indicates an angle of 45 degrees.

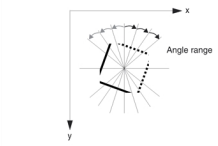
Setting the Angle and Angle Range

To compensate for differences in orientation between the model and the image, the Symbol tool can rotate the model before comparing it to the image, as shown in *Angle values and rotated Model* on page 456. The angular rotation can have any positive or negative value up to 180 degrees. In high-contrast operating mode, the Symbol tool automatically uses an angle range value of 180 degrees.



Angle values and rotated Model

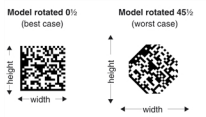
The Symbol tool can also rotate the model through a range of angles, as shown in *Angle range* on page 456.



Angle range

Searching an angle range reduces speed. In general, you should minimize rotational tolerance as much as possible.

If you change the angle of the model, you should increase the size of the region of interest. The exact increase depends on the angle, as shown in *Rotated model requires increased region of interest* on page 457.



Rotated model requires increased region of interest

In general, the region of interest must contain the entire symbol plus the following:

- A three-module space on all sides for the quiet zone
- An adequate positional tolerance
- A margin sufficient for rotating or rescaling the model

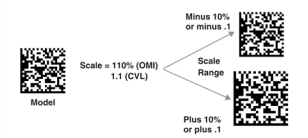
Setting the Scale and Scale Range

To compensate for differences in size between the model and the image, the Symbol tool can apply a scaling factor to the model before comparing it to the image. In high-contrast operating mode, the Symbol tool automatically uses a scale range value of 5 percent. *Scale values for a 2D symbol* on page 457 shows the result of the application of two scale factors.

Scale values for a 2D symbol

The Symbol tool can change the scale once and then use the scaled model in searching for symbols in the image.

The tool can also produce a set of rescaled models by multiplying the model by a range of positive and negative percentages. Then it uses each of the rescaled models in the search for a symbol.



Scale Range value, 2D symbol

If you apply a positive scale factor to the model, you should increase the size of the region of interest. The minimum size of the region of interest depends on the effective size of the model. When setting the region, include a margin adequate for the upper end of the scale range—including about three modules on all sides for the quiet zone.

Setting the Cell Size Range

The cell size range specifies the expected range of cell size in client units of the symbol to be learned. You should try to acquire images in which the cell size of the symbol falls within the default range. However, if you know in advance that the cell size varies substantially from the default range for the tool's operating mode, set the cell size range using the initial Learn parameters.

The optimal cell size range varies with operating mode and symbology. *Cell size range default values* on page 457 shows the default ranges for standard and high-contrast operating modes.

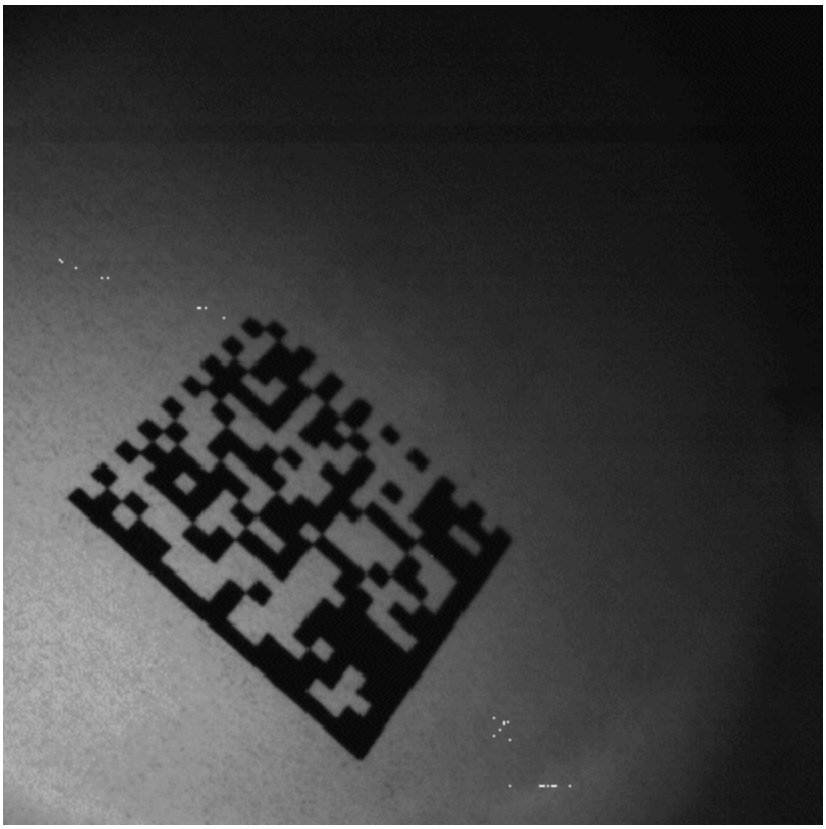
Mode	Symbol type	Range (min)	Range (max)
eStandard	QR code symbols	7 pixels	14 pixels
	Data matrix symbols	4.5 pixels	9 pixels
eHighContrast	Continuous symbols	7 pixels	21 pixels
	Dot-matrix symbols	7 pixels	14 pixels

Cell size range default values

Setting a cell size range larger than the default ranges can substantially increase the tool execution time during learning operations.

Perspective Distortion

The Symbol tool can compensate for any perspective distortion that might occur due to the spatial relationship between the symbol and the camera. For example, the following image shows a symbol exhibiting perspective distortion:

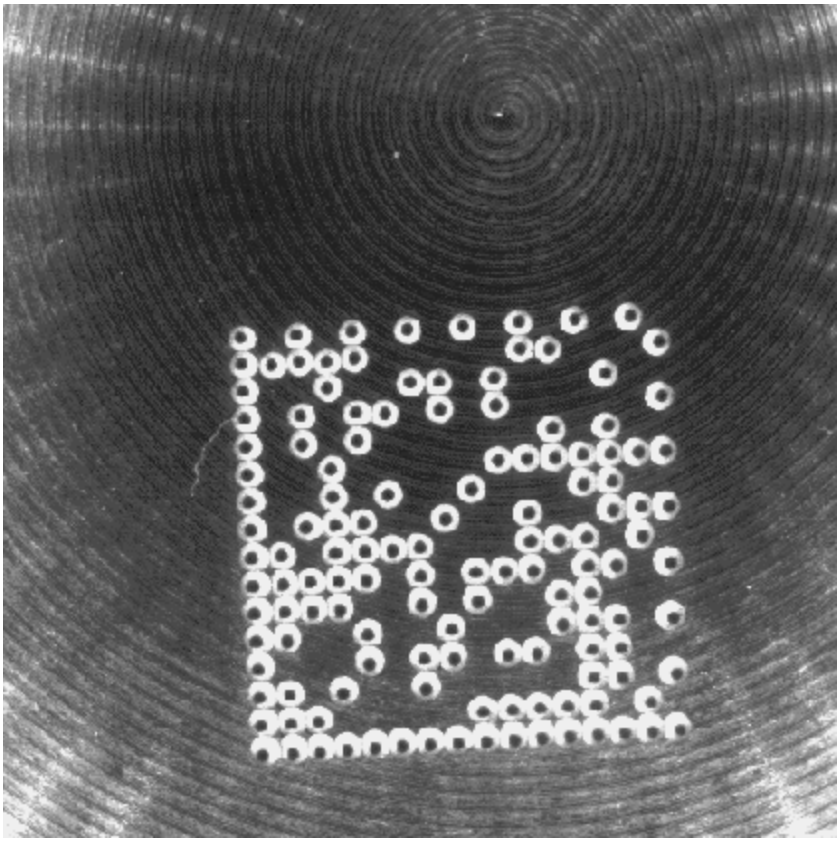


Perspective distortion

When you train a Symbol tool you can enable it to compensate for perspective distortion. In addition, you can enable or disable the feature as your vision application executes. Be aware, however, that enabling this feature can increase the average decoding time for all symbols. Enable the tool to handle perspective distortion only when it can occur in the images your application acquires.

Non-Conformant Modules

Images of a Data Matrix symbol might exhibit non-conformant modules, where each dot in a dot-peened matrix appears with both a bright and dark field. The following image shows a Data Matrix symbol exhibiting non-conformant modules:



Non-conformant modules

When you train a Symbol tool you can enable it to compensate for non-conformant modules. In addition, you can enable or disable the feature as your vision application executes.

This feature is available only for Data Matrix symbols.

Decoding a Symbol

Once the Symbol tool locates a symbol, it decodes it. Then it can return the raw bit stream encoded in the symbol as well as the ANSI string derived from the bitstream. If the tool is unable to decode the string, it returns an error code.

Symbol Tool Results

After locating and decoding a symbol, the Symbol tool reports the results summarized in *Results returned by the Symbol tool* on page 460:

Result	Description
Angle	The angle of the symbol with respect to the angle of the model
Aspect	The aspect ratio of the symbol that was found with respect to the client coordinate space
Decoded flag	True if the tool decodes the symbol
Decoded data	The raw bit-stream to be decoded as a sequence of bytes
Decoded multibyte character string	The decoded data as a null-terminated, multibyte character string.
Decoded string	The ANSI string decoded from the symbol

Result	Description
Found flag	True if the tool locates the symbol A result is found if its score equals or exceeds the acceptance threshold.
Image from client transform	The transform required to translate client coordinates to image coordinates
Learn parameters	The **ccAcuSymbolLearnParams** used to obtain these results
Location of the center of the symbol	A vector consisting of the following: • X-coordinate of the center of the symbol, which is the column offset from the left of the screen to the center • Y-coordinate of the center of the symbol, which is the row offset from the left of the screen to the center
Number of errors	The number of erroneous data words encountered while decoding the symbol
Number of error bits	The number of erroneous bits encountered while decoding the symbol
Result grid	The affine rectangle used to locate the symbol
Scale	The scale of the symbol expressed as a percentage of the model size
Score	The result score, a number between 0.0 and 1.0, where 1.0 indicates a perfect correlation between the symbol and the model If the score is 0.0 during the search phase, the tool does not perform the decoding step and sets angle, scale and aspect to their nominal values.
Time	The time required to decode the symbol

Results returned by the Symbol tool

The Symbol Tool

The Symbol tool operates in either of two modes:

1. *Learn and Decode* mode
2. *Standard Decode* mode

You determine which mode to use by calling a decode function for Standard Decode mode and a learn function for Learn and Decode mode.

Learn and Decode Mode

In this mode the tool operates in two phases: a learning phase and a decode phase.

During the learning phase, you provide a source image containing a symbol, and the tool locates the symbol using the Find parameters that you provide. It then estimates optimal values for those Learn parameters that you direct the tool to learn. You provide values for the other Learn parameters. The objective of the learning phase is to estimate the optimal values for a set of parameters when incomplete or partial information regarding those parameters is available. Based on the values of the Learn parameters that the tool discovers and that you provide, the tool creates a model of the symbol.

During the decode phase of Learn and Decode mode, the tool decodes the symbol and returns the decoded string and the results summarized in the section *Symbol Tool Results* on page 459.

Standard Decode Mode

In this mode, the tool operates only has a decode phase.The tool uses the model created during Learn and Decode mode and the values for Find parameters that you specify to locate the symbol. Then it decodes the symbol and returns the decoded string and the results summarized in the section *Symbol Tool Results* on page 459.

Developing an Application

This section presents basic information about developing an application.

Application Designs

The following list presents three ways that you can use the Symbol tool. How you use the tool depends on your application needs.

1. The application calls the learn function once on a typical image and then uses the decode function to decode all run-time symbols.

   For this application, conditions and parts remain relatively constant.

2. The application calls the learn function whenever the operator notices that conditions are changing. The application uses the decode function to decode the symbols.
3. The application calls the learn function to decode every symbol because of conditions may change at any time.

For most applications, you should call the learn function once for the following reasons:

- The Symbol tool can adjust to changing conditions as long as the symbol quality does not deteriorate beyond tolerable limits.
- The decode function is much faster than the learn function.

Obtaining a Clear Image of Your Part

To obtain a clear image of your part, perform the following steps:

- Adjust the camera and lens to obtain a clear, focused, high-contrast image. For good lighting and aspect response, place the camera perpendicular to the surface. Adjust the field-of-view so that each module of the symbol occupies at least a 5x5 pixel area. Insufficient resolution can cause failure to decode.
- Adjust the region of interest, making it as small as practical. Increasing the size of the region rapidly increases the number of pixels it contains and therefore increases processing time during the search phase. On the other hand, the region must contain the entire symbol (including a three-module space on all sides for the quiet zone) plus adequate positional tolerance (including a margin sufficient for rotating or rescaling the model during searching.
- If possible, acquire the image at a constant angle. The Symbol tool can rotate the model to compensate for angled images, but eliminating this adjustment saves time. In addition to the one-time adjustment, the Symbol tool can search a range of angles. Large ranges can incur a large computational cost, and limiting the range also saves processing time.
- If possible, acquire the image at a constant size. the Symbol tool can resize the model and can also search a range of sizes. The trade-offs are similar to those for the angular adjustments. Simple rescaling incurs a one-time cost. Searching a range incurs a higher cost that depends on the range.

Steps in Application Development

To write an application that uses the Symbol tool, you perform the following steps:

1. Write code to acquire images.
2. Construct a Symbol tool with an appropriate operating mode or use the default operating mode.
3. Specify those Learn parameters that you want the Symbol tool to learn.
4. Specify values for the other Learn parameters or use the defaults.
5. (Optional) Tune the lighting parameters.
6. Specify Find parameters.

7. Call the learn function to create the model and decode the string encoded in the symbol passed to the learn function.
8. Determine if learning has occurred.

   If the tool has learned the model, you can retrieve model parameters from an object containing the “best learned parameters.”
9. Call the decode function to decode all symbols for your application unless conditions change radically or you need to decode different symbols.
10. If you need the results of calling the decode function, you can retrieve them from a results object.

Optimizing the Symbol Tool’s Performance

You can optimize the learning and decoding abilities of the Symbol tool by following these guidelines:

- Select the most suitable operating mode for the tool.
- Obtain clear, focused, evenly-illuminated, high-contrast images.
- Select a region of interest that is as small as possible, while still satisfying the minimum quiet zone requirements.
- Set appropriate values for Learn and Find parameters whenever possible.
- Minimize the variation from learn to decode time.

Framework Differences

Framework differences between OMI and CVL on page 462 summarizes the differences between OMI and CVL:

Parameter	OMI	CVL
Acceptance threshold	The default is 30.	The default is 0.5.
Aspect ratio	The range is 500 to 2000. The default is 1023 for the default camera.	Range from 0.0 to 1.0.
Center of the symbol	The row and column offsets from the top left to the screen to the center.	X-coordinate of the center of the symbol is the column offset from the top left of the screen to the center. Y-coordinate of the center of the symbol is the row offset from the top left of the screen to the center.
Centering	Dynamically updates the scale and orientation of the model by resetting the angle and scale values to those of the last symbol found. Compensates for progressive change in angle or scale.	Not supported
Confusion threshold	The default is 70.	The default is 0.7.
Contrast	Range from 0 to 100%. Contrast threshold expressed as a percentage.	Range from 0.0 to 1.0. Contrast threshold expressed as a fraction.
Scale	Scale is a percentage of the model size. The scale factor can vary from 80 to 150 percent, with the original model being 100 percent.	Scale is a fraction of the model size and must be greater than 0.0.

Framework differences between OMI and CVL

Barcode Tool

Note: The functionality of this tool has largely been supplanted by the ID Tool. If you need to develop new code that reads 1D symbols, refer to the documentation for *ID Tool* on page 417.

The Barcode tool decodes AIM Code 39 and Code128, BC412, and IBM412 barcode formats. It also decodes PDF417 stacked barcodes. After decoding a barcode, the tool returns the raw and decoded strings, the score, the scan direction, and the decode time in seconds.

This chapter describes the Barcode tool and contains the following sections:

Some Useful Definitions on page 463 is a glossary of relevant terms.

Barcode Tool Overview on page 464 describes the Barcode tool's features.

How the Barcode Tool Works on page 465 describes the Barcode tool's operation.

Barcode Autocalibration on page 465 describes the barcode autocalibration feature.

Using the Barcode Tool on page 467 describes the techniques that you use to implement a Barcode application.

Framework Differences on page 471 summarizes the differences between the OMI and CVL versions of the Barcode tool.

PDF 417 Stacked Barcode on page 473 describes the PDF417 decoder. Although this is a separate tool from the Barcode tool, it is described in this chapter because PDF417 is a type of barcode.

Some Useful Definitions

acceptance threshold: The percentage of scan lines that the Barcode tool must correctly decode for the read operation to pass.

AIM USA: The trade association that publishes the Bc39 specification.

barcode: An array of vertical parallel lines and spaces (both called *bars*) used to represent data.

Bc39: A barcode symbology, also called Code 39 or Code 3-of-9, that supports all 36 alphanumeric characters, seven additional characters, a variable symbol length, and an optional checksum. Each character has nine elements (5 bars and 4 spaces), three of which are wide elements (hence "3 of 9").

Bc412: A barcode symbology that supports alphanumeric characters, A through Z, 0 through 9, dash (-), and the semichecksum characters 0 through 7 and A through H.

checksum: A character used in calculating a check for correct decoding. Passing checksum does not mean that the decode was successful. Apart from passing the checksum, the score should also exceed accept threshold. In CVL, score and the acceptance threshold range from 0 to 1.0. In OMI, the range for the acceptance threshold is from 0 to 100. The range for score is from 0 to 200.

Code128: A high-density barcode that can encode the entire ASCII character set (including control characters) as well as a packed decimal format that can store two digits in one byte. Each Code128 byte consists of three bars and three spaces.

darkfield: A kind of lighting that reflects light from the surface away from the lens, highlighting the characters against a dark background.

element: A single bar or space.

IBM412: A barcode symbology that supports alphanumeric characters, A through Z, 0 through 9, dash (-), and the semichecksum characters 0 through 7 and A through H.

light intensity: A tunable value controlling the overall brightness of the lights.

light mode: A tunable value controlling the balance of brightfield and darkfield lighting, which vary in an inverse relationship.

light tuning: An automatic operation that tests brightfield and darkfield light-intensity values, changing values through specified ranges by specified increments to find the best values. Used during setup to find initial values and in the final application to compensate for changes in the production environment.

PDF417: A multirow, variable-length 2D symbology that offers high data capacity and error correction capability.

percentage score: The number of scan lines divided by the number of scan lines that yield a string. This computation ignores blank lines and counts noisy lines only if they have the characteristics of a barcode. The tool compares the percentage score to the acceptance threshold to determine if the decoding operation was successful or not.

scan line: A vector that runs across the region of interest in an image and that is orthogonal to the barcode contained in the image. The Barcode tool analyzes the light-to-dark and dark-to-light transitions of the pixels that lie along this vector. If the tool finds a pattern of transitions that matches the pattern of the specified barcode, it decodes the string that is encoded by the pattern.

score: A result value that indicates the success or failure of the read operation. Any score greater than or equal to the acceptance threshold indicates success, and any score less than the threshold indicates failure.

symbol: An entire barcode, including the quiet zone, start-stop characters, data characters, and checksum.

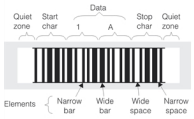
symbology: A set of rules that define a format for encoding and decoding barcodes.

UCC: The Uniform Code Council, the organization that publishes the UPC specification.

Barcode Tool Overview

A barcode is an array of vertical parallel lines and spaces (both called *bars*) used to represent data. The Barcode tool decodes barcodes of the following types: Bc39 (Code 39), Code128, BC412, IBM412.

Bc39 barcode symbol structure on page 464 shows a Bc39 barcode and identifies its parts.



Bc39 barcode symbol structure

The Barcode tool works by analyzing a series of vectors or *scan lines* that cross the image containing the barcode perpendicular to the barcode, as shown in *Scan lines crossing a barcode* on page 464.

Scan lines crossing a barcode

Note: The Barcode tool can scan barcodes in both right-to-left and left-to-right directions.

Along each scan line, the tool looks for patterns of dark-to-light and light-to-dark transitions. If a pattern meets certain criteria, the tool attempts to decode the bars represented by the pattern. After the tool has analyzed all scan lines, decoding those that fit the barcode pattern, it passes the decoding operation if the percent of correctly decoded lines exceeds an acceptance threshold, which you specify. The tool also calculates the checksum if it is enabled for the symbology. If the checksum is valid and the decoding operation has passed, the tool returns the raw and decoded strings, a score, the decode direction, and the amount of time required by the decoding operation.

Supported Barcode Symbologies

Supported symbologies on page 465 summarizes the characteristics of the symbologies supported by the Barcode Tool.

Type of Barcode	Field Positions	Max Chars	Legal Characters	Checksum
Bc39	Variable	45 per field	A through Z, 0 through 9, -, ., space, $, /, +, %.	Optional
Code 128	Variable	45 per field	All ASCII characters (printing and nonprinting)	Required
BC412	Variable up to 18	45 per field	A through Z, 0 through 9, -, any alphanumeric, semichecksum character (0 through 7, A through H)	Required
IBM412	Variable up to 18	45 per field	A through Z, 0 through 9, -, any alphanumeric, semichecksum character (0 through 7, A through H)	Required

Supported symbologies

For information about which symbologies are supported by the CVL and OMI frameworks, see the section *Framework Differences* on page 471.

For information about terminating a string, see the section *Marking the End of the Data Fields* on page 469.

How the Barcode Tool Works

To translate a barcode into its encoded string, the Barcode tool performs the following actions:

1. It scans the region of interest containing the barcode.
2. For each scan line, it decodes the detected bars and checks the following:
   - The start and stop codes
   - The checksum, if enabled
   - The character subset field by field

   A correct line must pass all three checks.
3. It computes the *percentage* score, which is the percentage of correct scan lines crossing the barcode compared to the total number of scan lines.
4. It compares the percentage score to the acceptance threshold. If the percentage score is greater than or equal to the acceptance threshold, the decoding operation is successful.

   If the score is below the acceptance threshold, the decoding operation fails.
5. It reports the result string, score, and decode time in seconds.

Barcode Autocalibration

CVL supports barcode autocalibration.

Runtime Inputs

In addition to a barcode image in one of the supported symbologies, the Barcode Calibration tool requires the following inputs:

- Vector of runtime parameters (**ccAcuBarCodeRunParams**) objects, which hold field strings for each position in the barcode, acceptance threshold, checksum flag, scan direction, light power, and dark level for each symbology to be calibrated.
- Result (**ccAcuBarCodeCalibrationResult**) object to store the calibration results.

API Changes

The following APIs implement barcode calibration:

- The Barcode Calibration tool is implemented as a member function of the **ccAcuBarCodeTool** class. This function is described in the *CVL Class Reference*.
- A **ccAcuBarCodeCalibrationResult** class stores the calibration results. This class is also described in the *CVL Class Reference*.

Operating Ranges

The properties of the input image must be within certain ranges to assure high performance of the calibration tool. A runtime image that conforms to the ranges shown in the following table is considered a reasonable image.

Input	Min	Max
Barcode height	15% of barcode length or 30 pixels, whichever is greater	- (no max limit)
Width of a narrow bar or space	3 pixels	10 pixels
Encoded string length	Code39: 1 character Code128: 1 character BC412: 2 characters IBM412: 2 characters	Code39: 20 characters Code128: 44 characters BC412: 18 characters IBM412: 18 characters
ROI	Must be large enough to enclose the entire barcode, including the leading and trailing Quiet Zone portions of the barcode. Minimum Quiet Zone width for Code39 and Code128 is 10 x-widths (where an x-width is the width of a narrow bar or space). Minimum Quiet Zone width for BC412 and IBM412 is 10 modules.	As small as practical. The barcode must occupy at least 20% of the height of the ROI.
Orientation (deviation from 0° or 180°)	0°	±3°

Calibration Results

Given a reasonable region of interest (ROI) in an image containing a barcode, the calibration can automatically detect and return the following properties of the barcode:

- Type of barcode
- Size, position, and pitch of barcode
- Length of encoded string (*Length of Encoded String* on page 467)
- Scan direction of barcode (*Scan Direction* on page 467)

The calibration produces the following in software:

- Client from image transform (**cc2Xform** object)
- Time (execution time of the calibration process)

After a calibration, the **ccAcuBarCodeCalibrationResult** object contains the information obtained from calibration (shown in the list above) as well as the results of the decode operation performed using the calibrated barcode properties (shown in the list below):

- Pass/fail flag
- Score
- Decoded string
- Raw data
- Time (execution time of the decode process)
- Checksum validity flag

Type of Barcode

By examining the pattern of bars in the image, autocalibration determines whether the symbology represents a type Code39, Code128, BC412, or IBM412 barcode.

Size, Position, and Pitch of Barcode

Autocalibration determines the size, position, and pitch of the barcode in terms of a rectangle that best fits the limits of the barcode. The size of the barcode is the height and width in pixels of the best fit rectangle. The position of the barcode is the center of the best fit rectangle. The pitch is defined as (2*w/n), where w is the width of the rectangle and n is the number of edges in the barcode.

To determine the barcode size, the calibration analyzes a series of vectors, or scan lines, across the image, perpendicular to the bars. Along each scan line, the tool searches for patterns of dark-to-light or light-to-dark transitions. If the pattern meets the criteria of a particular symbology, the scan line is considered valid. The tool then attempts to decode the string represented by the pattern.

The barcode height is determined by the set of valid scan lines defining the upper and lower sides of the rectangle that specifies the limits of the barcode. As a consequence, the returned barcode height is only an approximation. You can improve the accuracy of the returned barcode height by making the ROI as small as practical.

The barcode width is calculated as the average of the distance between the first and last edges of the barcode along the valid scan lines.

Length of Encoded String

Autocalibration determines the length of the string encoded by the barcode. The string length is the number of characters encoded by the barcode including the checksum character, which is optional for Code39 and mandatory for BC412 and IBM412 barcodes. While mandatory for Code128, the checksum character is not present in the string returned by Code128 barcodes. Hence, for Code128 the length of the encoded string does not include the checksum characters.

Scan Direction

Autocalibration determines whether the scan direction is 0° (left to right) or 180° (right to left). The scan direction is that orientation of the barcode that yields the correct string.

Using the Barcode Tool

This section describes how to use the Barcode tool.

Guidelines for Acquiring Images

This section lists the steps that you can take to maximize the quality of the images that your application inspects.

- Place the camera perpendicular to the surface of the part so that the image of the barcode is not skewed.
- Adjust the camera and lens to obtain a clear, focused, high-contrast image.

  The image should be big enough so that the narrowest bar or space is at least three pixels across.
- Ensure that the region of interest contains the entire barcode.

- Increase the number of scan lines that cross the entire barcode by reducing the amount of skew or increasing the height of the barcode, or both.
- Make the region of interest as small as practical.

  To assure that enough scan lines cross the barcode image, the barcode must occupy at least 20% of the total image height.

  Strong vertical, non-barcode lines located inside the region of interest on either side of the barcode can cause a failure to decode.
- Ensure that the image includes the leading and trailing quiet zone portions of the barcode.

  The minimum quiet zone width for BC412 and IBM412 symbologies is 10 modules, where a module is the width of one bar or one space.

  The minimum quiet zone width for the Bc39 symbology is the greater of 10 x-widths (where an x-width is equal to the width of a narrow bar or space) or 0.10 inches.

  The minimum quiet zone width for the Code128 symbology is the greater of 10 x-widths (where an x-width is equal to the width of the narrowest bar or space) or 0.075 inches.

Acceptable image of a Bc39 barcode on page 468 shows an example of an acceptable image containing a Bc39 barcode.

Acceptable image of a Bc39 barcode

Specifying Barcode Symbology

Barcode symbologies differ in the number of field positions, the type of characters legal at each position, and the presence or absence of a checksum. To use the Barcode tool, you must specify the symbology of the barcodes that your application will decode.

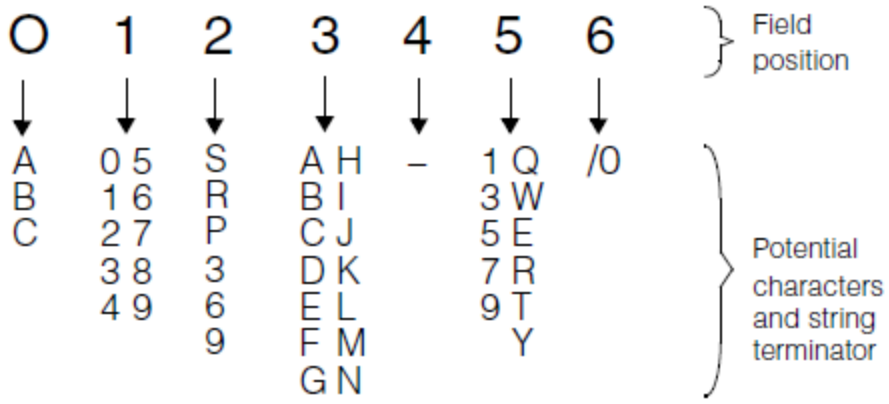
For information about how to specify a symbology in the OMI and CVL frameworks, see the section *Framework Differences* on page 471.

Defining the Permissible Characters for Each Field

After defining the symbology of the barcodes that your application will decode, you specify the following information:

- The number of field positions in the barcode
- The character values that the tool should look for in each field

Constraining each field to a known set of characters, as shown in *Barcode field positions limited to character subsets* on page 469, improves speed and reliability.



Barcode field positions limited to character subsets

Marking the End of the Data Fields

In the field that follows the last field position that contains data, you enter \0 to mark the end of the encoded string. The Barcode tool requires this termination character and ignores any subsequent fields. Specifying the wrong number of fields prevents the Barcode tool from decoding the barcode.

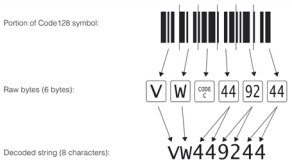
Note that the maximum number of characters that you can specify is one more than the maximum number of characters that a barcode can contain. This allows you to specify an end of string character for a maximum-length string.

Note: In each line, the Barcode tool compares each character actually decoded with the list of characters that are legal for that field position. If a decoded character is absent from the list of legal characters, then the decode for that field fails.

Code128 Considerations

Code128 is unique among the symbologies supported by the Barcode tool in that it uses shift codes to switch between multiple character sets. For example, Code Set C uses values 0 through 99 to represent the 100 possible pairs of digits. An entirely numeric barcode might start with the Start C code, while an alphanumeric barcode might use the CODE C and CODE A codes to switch in and out the Code Set C. This use of shift codes means that the number of raw bytes in a barcode will likely be different than the number of characters in the decoded symbol.

Decoding a Code128 symbol on page 469 shows an example of how the number of raw bytes in a Code128 symbol may differ from the number of characters in the decoded string.



Decoding a Code128 symbol

If you specify field codes for a Code128 symbol, you should match the field codes with the decoded string as you would for other barcodes. It does not matter where the control characters are placed in the barcode. However, best results are obtained by not specifying fielding for Code128. This is true for two reasons:

- Code128 is used only in label applications where there is little or no confusion in the barcode region. Hence, the Barcode tool can easily determine the length of the string and the encoded characters.
- In Code128 symbology, a single string can be split across multiple barcodes by using a certain control character. In such cases, specifying fielding to indicate the length of the encoded string can be meaningless.

Tuning Light Intensity Values

For all supported barcodes, the Barcode tool can evaluate *brightfield* and *darkfield* light-intensity values to find those best suited for recognizing the barcode. These values control the balance of brightfield and darkfield lighting, which vary in an inverse relationship. Brightfield lighting reflects light from the surface into the lens, so characters tend to be dark on a bright background. Darkfield lighting reflects light from the surface away from the lens, highlighting characters against a dark background

The tool tests a range of light-intensity values to find the best values. You can use this facility during setup to find initial values and in the final application to compensate for changes in the production environment.

Setting the Acceptance Threshold

The *acceptance threshold* is the percentage of scan lines crossing the barcode that must correctly decode for the decoding operation to pass. The tool compares the acceptance threshold to the percentage score to determine if the decoding operation was successful or not. *Barcode acceptance threshold* on page 470 illustrates the role of the acceptance threshold in determining success or failure.

Barcode acceptance threshold

How the Tool Determines the Percentage Score

After scanning the image and attempting to decode each scan line, the Barcode tool calculates the percentage score by dividing the number of scan lines that yielded a decode string by the total number of scan lines. This computation ignores blank lines and counts noisy lines only if they have the characteristics of a barcode. Setting the acceptance threshold to different values enables you accept or reject the decoding operation based the degree of confidence required by your application.

Setting a Low Acceptance Threshold

You might set a low threshold under the following circumstances:

- Some scan lines fail because they pass through an unclear section of the barcode image while other scan lines cross through clear regions and decode correctly. Enough scan lines are decoded correctly to ensure a correctly decoded string.
- Correctly decoding even one scan line provides sufficient reliability, especially if the checksum passes.

Setting a High Acceptance Threshold

You might set a high threshold under the following circumstances:

- The application is likely to encounter noisy or distorted images.
- The application must ensure that no scan line that crosses a barcode can be misread unless the image of the barcode yields no information.

Enabling the Checksum Calculation

For barcode formats with an optional checksum, you should enable or disable the checksum option to agree with the barcode. For barcode formats with a required checksum, checksum is always enabled.

For Bc39, BC412, and IBM412 barcodes, the Barcode tool reports both the data and checksum fields. For Code128 barcodes, the tool reports only the data fields.

Getting the Results

After a decode operation, the Barcode tool returns the following results:

- The result string

  For Bc39 barcodes, the Barcode tool reports both the data and checksum fields. For IBM412, and BC412 barcodes, it reports only the data fields.

  - On success, the result string contains the decoded characters.
  - On failure, if the barcode uses a checksum, then all fields contain asterisks. If the barcode does not use a checksum, then only the field positions that could not be decoded are represented with asterisks.

- The raw string

  For Code128 barcodes, the raw string includes the actual bytes read from the barcode. These bytes will include shift codes and two-digit packed bytes but not the start and stop codes. For barcodes other than Code128, the raw string is the same as the decoded string.

- The direction of the decode

  All barcodes can be decoded left-to-right or right-to-left. You do not need to determine the orientation of the code before reading; the tool will tell you the direction in which the code was read.

- The score value
- The decode time in seconds

Framework Differences

This section summarizes the differences between the OMI and CVL versions of the Barcode tool.

Supported Symbologies

CVL: Bc39 (Code 39), Code128, BC412, IBM412

OMI: Bc39, BC412, IBM412

Methods of Specifying Symbologies in CVL and OMI

The table below summarizes the methods used for specifying symbology in OMI and CVL:

Type of Barcode	OMI GUI	OMI API	CVL API	Specifying Checksum
BC412 or IBM412	Set **CodeType** field in dialog box.	Use **SetCodeType()** function.	Construct a **ccAcuBarCodeRunParams** object with the symbol type specified as an argument.	Checksum always enabled. No dialog box field for OMI.
Code 128	Same as above	Same as above	Construct a **ccAcuBarCodeRunParams** object with the symbol type specified as an argument.	Checksum always enabled.
Bc39	Set **CodeType** field in dialog box.	Use **SetCodeType()** function.	Construct a **ccAcuBarCodeRunParams** object with the symbol type specified as an argument.	Optionally check **CheckDigit** control in OMI dialog box for or use API function **SetCheckDigit()**. Enable using **checksum ()** method of the run-params object.

Setting the barcode symbology in OMI and CVL

Defining the Permissible Characters for Each Field

In OMI, when you create a Barcode tool, all the field positions contain the same set of characters. You can then configure the field to contain a type of character or specific characters. In CVL, when you create a Barcode tool, all fields are initialized to the type that you specify at construction time. Then, using the different setters, you can either set fields to contain a certain character type or specific characters.

Note: For Code128 symbols, Cognex recommends that you specify that any field position may contain any character.

Scores in OMI and CVL

This section contrasts the scores obtained by the OMI and CVL versions of the tool.

OMI Scores

The table below summarizes the various scores that the OMI version of the Barcode tool can generate and ways of interpreting the scores:

Item	Score	Explanation
Acceptance threshold	Maximum: 100 Minimum: 0	Pass only if every scan line that crosses a barcode decodes and checksum (if enabled) is correct. Report results regardless of number of fields decoded.
Checksum	Amount added to score if correct: 100 Score if invalid: 0	Valid checksum always yields a successful score. Failing the checksum always yields a 0 score.
Failure		All failures produce a score of 0.
Maximum possible score	200	• Decoded string contains a correct checksum. • The checksum option is enabled. • All scan lines that cross the barcode decode correctly.
Success	Greater than or equal to the acceptance threshold	If a barcode checksum is disabled, then only acceptance threshold controls success or failure.

Interpreting the scores returned by the OMI version of the Barcode tool

The following are examples of various scores and ways of interpreting them:

- If the acceptance threshold is 0 and the checksum passes, then the score is 100 for the checksum plus the percentage of scan lines that decode correctly. (success)
- If the threshold is 70, 60% of the scan lines decode correctly, and the checksum passes, then the score is 0 (failure).
- If the threshold is 50, 60% of the scan lines decode correctly, and the checksum fails, then the score is 0. (failure)
- If there is no checksum and the threshold is 50, then 60 passes and scores 60, and 40 fails and scores 0.

For IBM412, and BC412 barcodes, which always have a checksum, the possible scores are 0, 100, or the sum of 100 plus the percentage of scan lines that decode. For Bc39 barcodes, which have an optional checksum, the possible scores depend on the acceptance threshold and on whether the checksum is enabled.

CVL Scores

The table below summarizes the scores that the CVL version of the Barcode tool can generate:

Item	Score	Comment
Acceptance threshold	Maximum: 1.0 Minimum: 0.0	Barcode image must generate a score that equals or exceeds the acceptance threshold.

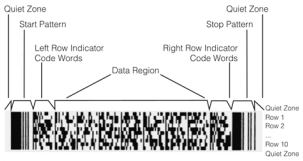
Item	Score	Comment
Checksum	true or false	The *checksumValid* data member of **ccAcuBarCodeResult** is set to true if the checksum is correct. The *checksumValid* data member of **ccAcuBarCodeResult** is set to false if the checksum is invalid.
Success	Greater than or equal to the acceptance threshold. You can also test *checksumValid* if checksum is enabled.	If checksum is disabled, then acceptance threshold controls success or failure.

Interpreting the scores returned by the CVL version of the Barcode tool

PDF 417 Stacked Barcode

The CVL ID tools include a PDF417 decoder. PDF417 is a multirow, variable-length 2D symbology that offers high data capacity and error correction capability. The current tool expects the symbol to be the predominant feature in the image.

The following is an example of a typical PDF417 symbol and its structure.



PDF417 Symbol Structure

Every PDF417 symbol contains a minimum of 3 rows to a maximum of 90 rows. Each row consists of the following elements:

1. Leading Quiet zone
2. Start pattern
3. Left row indicator symbol character
4. One to thirty data symbol characters
5. Right row indicator symbol character
6. Stop pattern
7. Trailing Quiet zone

A symbol character consists of seventeen modules arranged into four bars and four spaces. Each symbol character represents a value in the range of 0 to 928. These symbol character values are often referred to as *codewords* (not to be confused with the term codewords used in error correction literature).

Because the number of rows is variable, and rows are of variable length in the number of symbol character columns they contain, the height/width proportion, or aspect ratio, of a PDF417 symbol can be varied to suit spatial requirements for printing. However, the number of symbol characters in all rows of a given symbol must be the same.

The data region of a PDF417 symbol is the central area of codeword columns between the left row indicator column and the right row indicator column. The first (upper left) codeword of the data region is the symbol length descriptor. Its value indicates the total number of codewords in the data region, including the symbol length descriptor itself, but excluding the error correction codewords. The remaining codewords in the data region (including the data codewords, pad codewords, and error correction codewords, in that order) are arranged with the most significant codeword adjacent to the symbol length descriptor, and are read from left to right, top to bottom. The total number of codewords in the data region of a single PDF417 symbol cannot exceed 928.

Symbol Character Encoding

Symbol Character Structure

The symbol character consists of four bars and four spaces, each of which contains one to six modules. In all cases, the four bars and four spaces of any symbol character measure 17 modules in total. The width of one module is the X dimension of that symbol.

Clusters and Symbol Character Definitions

The entire set of PDF417 symbol characters is divided into three mutually exclusive clusters. Each cluster encodes all 929 defined PDF417 codewords with distinct bar and space patterns. Within each cluster, each symbol character is associated with a unique value in the range of 0 to 928. This value is the symbol character value or codeword.

PDF417 APIs

The following two classes and one global function comprise the PDF417 symbol tool:

- **ccPDF417Result**: Main class that holds the result returned by a PDF417 decoder.
- **ccPDF417Defs**: Namespace in which to declare errors related to PDF417 symbol decoding.
- **cfPDF417Decode()**: Global function that invokes the decoding process.

Input Ranges

The properties of the input image must be within certain ranges to assure high performance of the decoder. A runtime image that conforms to the ranges shown in the following table is considered a reasonable image.

Input	Requirement (Typical)
Image size (W x H)	640 x 480 pixels
X-Dimension	1.5 pixels (min)
Symbol rotation Rotation is the direction, or orientation, of the PDF417 symbol. It is defined by a vector running across the region of interest in an image and indicates the symbol's deviation from 0°. Symbol rotation is orthogonal to the bars in the PDF417 symbol from the Start pattern to the Stop pattern.	±5° (max) if the number of data columns is less than 4. ±2° (max) if the number of data columns is between 4 and 10.
Background	Clean and flat (such as a label placed on a web)
Contrast	>= 15 gray levels
Quiet Zone The quiet zone is a region of white space. The image must have quiet zones at least 2X wide on each side of the symbol, where X is the width of the narrowest bar.	2X (min)

OCR Tool

The OCR tool performs *optical character recognition* (OCR), a process for reading character strings.

Some Useful Definitions on page 475 defines some terms you will encounter in this chapter.

OCR Tool Overview on page 475 outlines how the OCR tool works.

Using the OCR Tool on page 486 describes the techniques that you use to implement an OCR application.

OCR Font on page 495 describes the OCR Font class, which is a container class for a font.

Some Useful Definitions

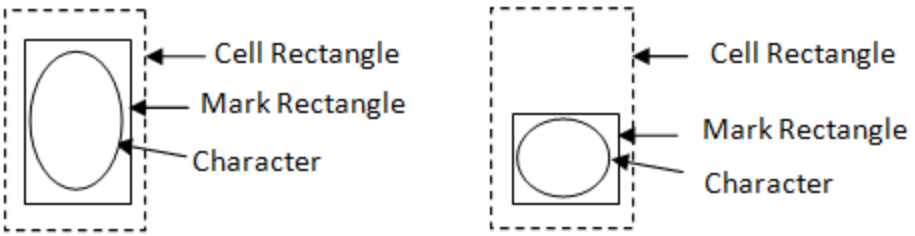
This section defines some terms and concepts used in this chapter.

affine rectangle: A quadrilateral where the opposite sides are parallel to each other

mark rectangle: A rectangle that specifies the x and y extents of a character in the rectified image.

cell rectangle: A rectangle that is in the same image coordinates as the rectified image and that provides information about the mark rectangle's position with respect to the read line. The cell rectangle can be used for discriminating characters with different sizes/positions (such as capital O versus lower-case o) based on character sizes with respect to other character sizes in the string.

See the following figure for mark rectangles and cell rectangles.



pitch: The distance between (approximately) corresponding points on adjacent characters.

Note: It is not the distance from the end of one character to the beginning of the next character (which is called the inter-character gap).

rotation: The turning of an object or rectangle about an axis point or center.

skew: The distortion of a rectangle into a parallelogram.

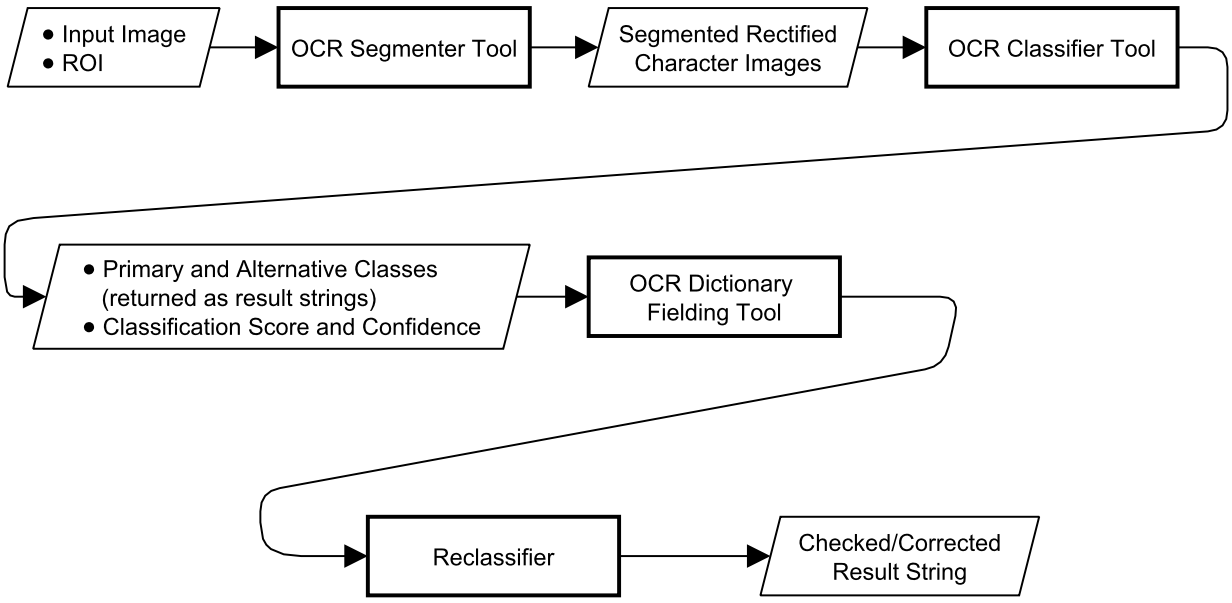
OCR Tool Overview

The OCR tool comprises the following components:

1. OCR Segmenter tool, performing the following:
    a. Image preprocessing and rectification
    b. Image binarization
    c. Character segmentation
2. OCR Classifier tool, performing classification of segmented rectified images.
3. OCR Dictionary Fielding tool, performing string verification and correction.

Note: To perform optical character recognition, using the OCR Segmenter tool and the OCR Classifier tool are required steps, whereas using the OCR Dictionary Fielding tool is an optional step.

The following figure illustrates the OCR process.



OCR process

The OCR Segmenter tool takes an input image and the region of interest (ROI) that contains the string to be read in the input image and outputs segmented rectified images of the characters in the string, which become classified by the OCR Classifier tool. The output from the OCR Classifier tool is passed to the OCR Dictionary Fielding tool, which determines the position of the fielding vis-a-vis the read string. Lastly, the characters are reclassified based on the fielding information for each character position.

OCR Segmenter Tool

The OCR Segmenter tool takes an input image and the ROI that contains a single line of characters and the surrounding featureless background (although the background might be non-uniform and/or noisy) and splits it up into individual regions where each region contains one character. The ROI is specified as an affine rectangle that specifies the approximate location, angle, and skew of the line of text. The ROI must be entirely contained within the input image.

For example, if an image contains the line of text “ABCD”, then the OCR Segmenter tool produces four separate images, each of which is an image of a single character. Character segmentation does not know anything about actually determining which letter a character image corresponds to; in the above example, it has no way of knowing that the first image corresponds to the letter “A”.

The OCR Segmenter tool is defined in the *ch_cvl/ocsegmnt.h* header file.

The tool determines where the top and bottom of the line of characters are and is capable of refining the angle and skew of the line if requested.

The x-axis of the affine rectangle should be parallel to the baseline of the text, and the y-axis should be parallel to the vertical strokes of the characters; if there is no skew, the y-axis is perpendicular to the x-axis. The positive direction of the x-axis corresponds to the reading direction. The positive direction of the y-axis corresponds to the vector from the top of each character to the bottom of each character. Note that the baseline of the text may have any angle 0-360 in the image as long as the affine rectangle is oriented at approximately the same angle.

For detailed explanations on affine rectangle and its transformations such as skew and rotation, see section *Affine Rectangle Sampling Tool* on page 110.

Note: The OCR Segmenter tool is not a general purpose string finder; it is not capable of finding a string in an arbitrarily complex image with a large ROI.

The OCR Segmenter tool performs the following functionalities on the input image in the following order.

1. Image Preprocessing

The OCR Segmenter tool has built-in preprocessing that handles a wide variety of images. For each analysis mode, the preprocessing parameters are set to fixed values and cannot be modified. The only way to vary the preprocessing is to change the analysis mode.

2. Image Binarization

Binarization takes the preprocessed image and an ROI and outputs a binarized image and ROI. Each pixel is labeled as either foreground (that is, potentially part of a character) or background. Labeling is performed based on a threshold level that is expressed as a probability value.

Note: Binarization typically has no information about characters or fonts.

3. Character Segmentation

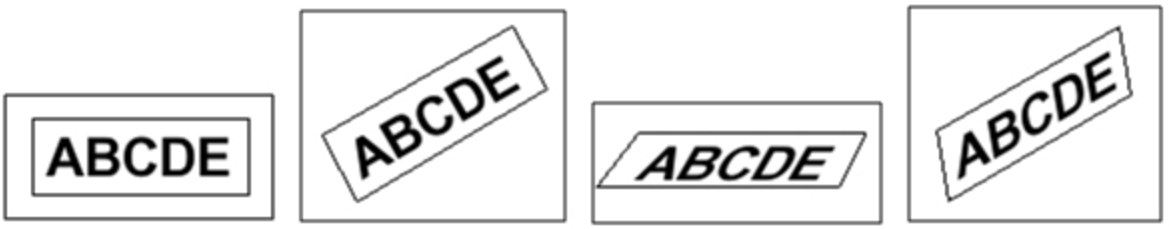
After input image preprocessing and binarization, character segmentation is performed by the OCR Segmenter tool.

In the ROI, each character is expected to occupy a single horizontally contiguous range, with no other character vertically above or below it. Good characters are expected to not have interior large horizontal gaps; with the exception that dot-matrix characters will exhibit expected intracharacter gaps. In other words, a line of characters should have left-to-right or right-to-left layout, which is typical of most Western languages. Characters of other languages are in general supported, although languages with unusual characters or layout may not work. For example, languages where a single character may occur as two or more horizontally separate pieces may not work, languages where two or more characters may be stacked on top of each other may not work, and so on.

The OCR Segmenter tool can (without a separate string finder) handle essentially arbitrary character sizes (for exact limitations, see *OCR Segmenter Tool Limitations* on page 478), rotation of ±15 degrees or more, and a large amount of translation, with the restriction that the ROI must contain only characters and background without other significant features (such as other lines of characters, label edges, and so on). For clean backgrounds, the ROI can essentially be arbitrarily large; for significantly noisier backgrounds, the ROI needs to be tighter around the characters.

The OCR Segmenter tool requires that there be at least a few pixels of background visible on all sides of the line of characters; in particular, the top, bottom, left, and right edges of the line of characters are not allowed to touch the edge of the ROI. The tool may completely fail (for example, find no characters or find completely wrong characters) unless the search region includes background pels on all sides of the characters. A general rule of thumb is that a surrounding border of about half a character size is usually ideal, although it can be less or substantially more as mentioned previously.

The figure below shows good ROI specifications for lines with and without rotation and skew. The outer rectangle is the image boundary, and the inner affine rectangle is the ROI.



Good ROIs

The figure below shows bad ROI specifications.

Bad ROIs

The OCR Segmenter tool can handle significant background gradients and noise. The main limitation is that the line of characters needs to be at least locally binarizable (that is, the strokes of a character need to be either all darker or all lighter than the background surrounding those strokes) and all the characters in a line need to be of the same polarity (that is, all dark-on-light or all light-on-dark). In most cases, the tool automatically detects the polarity of a line of characters. In some cases with very noisy backgrounds, you might need to specify the polarity. Also, specifying the polarity will decrease execution time.

The OCR Segmenter tool can handle broken characters (for example, a stroke character that is split into two or more pieces), although doing so may require that you perform parameter adjustments. The tool can optionally produce space characters in the gaps between segmented characters using some parameters you can specify to indicate how space insertion should be handled.

Character Segmentation Parameters

The OCR Segmenter tool provides a number of parameters you can control. Although in many cases the default values work well, in other (typically more challenging) cases, you might need to specify some parameter values. Most typically, you might need to specify a minimum character width, a maximum width, and/or a minimum pitch (character-to-character distance; for example, left edge to left edge). A *minimum fragment size* parameter is also necessary sometimes to handle noisy images or fonts with small characters such as periods.

The OCR Segmenter tool shows you the binarized/segmented image for diagnostic purposes to help you understand and fix problematic cases.

OCR Segmenter Tool Limitations

The following limitations apply to the OCR Segmenter tool:

- Backgrounds with strong textures and/or with so much noise that the character blends into the background may not be handled well.
- The ROI must not contain strong features other than the line of characters (for example, no other lines of characters, no label edges, and so on). In some cases, this may require precise fixturing. The OCR Segmenter tool can sometimes be made to reject other strong features using the parameters *ignoreBorderFragments* and/or *characterMaxHeight*.
- Touching characters can be handled to some degree but typically require you to adjust some parameters. Fixed-width touching characters can usually be handled by specifying the width; however, proportional fonts with touching characters are problematic, the OCR Segmenter tool may handle some cases correctly, but there may be some cases which cannot be handled correctly by the OCR Segmenter tool.
- Extraneous scratches or strokes (such as handwritten scribbles that might pass through characters on a bank check) may not be handleable by the OCR Segmenter tool.
- Currently, no template-based segmenter is available. Such a segmenter may be able to handle touching characters and extraneous strokes, but it would have other limitations.

- Line rotation can be determined, but for very short lines (for example, three or fewer characters) or relatively short lines with a lot of line jitter, it may be necessary for you to specify the rotation instead (because of the inherent uncertainty in determining the orientation of a short line).
- All characters in one line of characters must have the same rotation.
- All characters in one line of characters must have the same skew.
- For well-separated dot-matrix print (that is, where the dots are not touching), it is more likely that you will need to specify some parameters (such as expected character size and/or minimum inter-character gap) to get a good segmentation.
- *Don't-care* masks are not supported.
- Nonlinear client coordinates are not supported.
- Image size is limited only by physical memory (for example, images greater than 2 GB are supported on 64-bit systems).
- The stroke width must be greater than or equal to two pixels. For the definition of stroke width, see section *Font Stroke Width* on page 441.
- The minimum character size is 8x8 pixels for large (typically alphanumeric) characters and 2x2 pixels for small characters (such as periods).

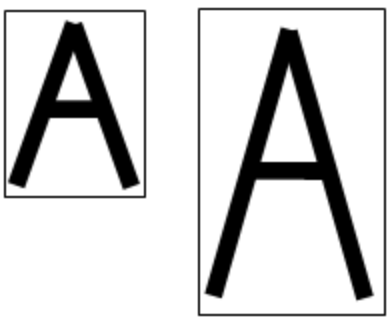
OCR Classifier Tool

The OCR Classifier tool is a trainable/runnable classifier that classifies segmented rectified images provided by the OCR Segmenter tool. (The OCR Classifier tool is defined in *ch_cvl/ocrclass.h*.) The OCR Classifier tool is trained from a collection of rectified training images (which in this case mean segmented regions), and then runs on a rectified run-time image of a character (which corresponds to the segmented region) and provides the classification by returning the best matching character (primary class) and the score and confidence of that best matching character.

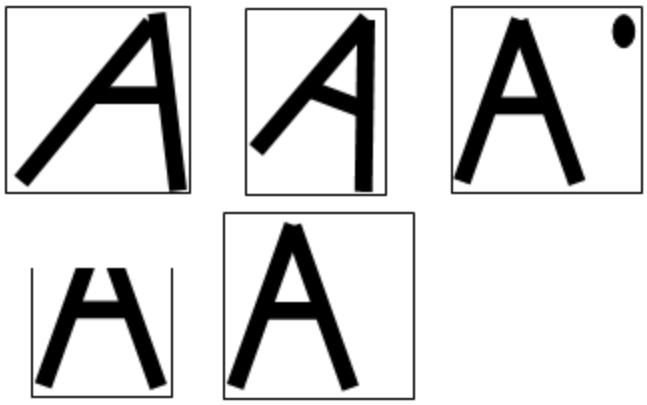
The OCR Classifier tool analyzes each rectified image's character cell rectangle (image coordinate pixel rectangle) with respect to the rectified image's character mark rectangle. The classifier algorithm expects the mark rectangle to be relatively accurate and it expects the cell rectangle to be less accurate. Consequently, the classifier algorithm, for the most part, ignores the pixels in between the rectified image's character cell rectangle and the rectified image's mark rectangle. The classifier algorithm mainly considers the size and position of the cell rectangle window. The classifier algorithm requires that the cell rectangle be specified for every train-time/run-time character.

The rectified run-time character images do not have to be exactly the same as the rectified train-time character images. In other words, the OCR Classifier tool should handle slight rotation and skew, arbitrary x-scale and y-scale, and reasonable variation in terms of character texture and background texture. The rectified image should contain a single character at the appropriate orientation, and the rectified image should be tight around the character's mark rectangle.

The following figures show good and bad rectified images surrounded by their mark rectangles.



Good rectified images



Bad rectified images

The rectified image passed to the OCR Classifier tool should include the pels in the mark rectangle as well as pels around the mark rectangle (that is, padding) in case the tool wants to look outside the given mark rectangle. Note that the rectified image includes only the mark rectangle, but the entire image of the rectified image includes padding pels.

The OCR Classifier tool is trained from one or more examples of each character to be classified. The tool determines the class (character code) of the rectified input image by finding the class of training instances that best match the rectified run-time input image; and it can optionally indicate which training instance matched. The tool can be incrementally trained with additional training images at any time. See *Motivation for Multiple Training Instances* on page 482.

The OCR Classifier tool can discriminate between characters with more significant degradation if the accept threshold is low enough and there are few character classes in the font. As a general rule, the more similar the classes in the font are, the less degradation the tool can tolerate while still consistently correctly classifying characters. The tool works on connected stroke characters, disconnected stroke characters, and dot-matrix characters. The tool is able to disambiguate characters based on size and cell rectangle (especially if they are large differences, such as capital O versus lower-case o).

The tool expects the rectified character images to have bimodal histograms, see the following figure for illustration.

Images with different types of histograms

The tool ignores where the rectified image came from with regards to the original image; for example, the tool ignores whether the rectified image comes from a region touching the image boundary.

The OCR Classifier tool is an image-coordinate tool. It ignores the client coordinates in the rectified images.

The OCR Classifier tool trains and runs on **ccOCChar** objects (see *ch_cvl/occhar.h*). Each **ccOCChar** object contains a rectified image as well as a 32-bit character code. The **ccOCChar** object contains additional information as well, including instance IDs and alphabet IDs, but the tool only considers the rectified image, its mark rectangle, its cell rectangle, and the 32-bit character code.

The OCR Classifier tool returns at most 1 instance per character code (that is, if 4 Z's scored highest, at most 1 of the Z's is returned, which is the highest scoring Z). Some algorithms of the OCR Classifier tool may optionally return specific training instances.

The OCR Classifier tool supports:

- Single-channel greyscale images; 3-channel color images are not supported.
- Timeouts with latency of 1 ms (CVL standard).

Handling Scale Variation by Resampling to Fixed Size

The OCR Classifier tool is designed to handle x-scale/y-scale/uniform-scale/aspect variations in the run-time character in the rectified image. The tool always resamples each rectified image to a predetermined size (the same resampled size is used for both train-time rectified images and run-time rectified images).

Note that the OCR Classifier tool can be set to rescale while maintaining the rectified image's aspect ratio. In this case, each character can be resampled to a different width, although they will all be resampled to the same height.

Confidence and Confusion

The OCR Classifier tool determines not only the classification (character code/instance) of the run-time rectified image, but it also reports the score of that classification and the confidence of that classification. The score is an indication of the closeness of the match to the training instances. The confidence is computed as the difference between the score of the classification (the highest scoring training instance) and the score of the next highest classification (the highest scoring training instance from a different class). The OCR Classifier tool's result includes a status (READ, CONFUSED, or FAILED) indicating the quality of the result.

The OCR Classifier tool performs *internal classification validation checks* to verify/validate that the highest scoring candidate class is the correct classification. If this validation fails, the highest scoring candidate is marked as CONFUSED and the confidence score is set to 0. Some examples of internal classification validation checks involve rescoring the candidates using different metrics. The correct match will always score highest regardless of the match metric. The result of this validation does not affect the result score.

In addition to determining the classification, the score, and the confidence score, the OCR Classifier tool also reports a set of alternative classifications. The alternative classifications are all of the classes that induce sufficiently high scores. The confusion character is defined to be the highest scoring alternative character that is not a swap character of the highest scoring character. (For the definition of swap characters, see section *Swap Characters* on page 484.) There will always be at least one alternative character/confusion character so long as the highest scoring character met the accept threshold and so long as there is at least one other (non-swap) class with non-zero score.

Note: The alternative classifications are computed in a more sophisticated manner than simply finding all classifications that induce scores satisfying the accept threshold. Instead, the alternative classifications are computed as any class that induced a non-zero score satisfying the following alternative score threshold:

[(the lowest score that is greater than or equal to the accept threshold) – (confidence threshold)] + (one different character more than that)

The alternative characters are sorted in the order of decreasing score.

The status is:

- READ if its score satisfies the accept threshold and the confidence score satisfies the confidence threshold.

Note: Satisfying the accept threshold means that the score is greater than or equal to the *acceptThreshold*, and satisfying the confidence threshold means that the *confidenceScore* is greater than or equal to the *confidenceThreshold*.

- CONFUSED if the score satisfies the accept threshold but either the confidence score does not satisfy the confidence threshold or an internal classification validation check does not pass.

  When the result is CONFUSED, the **confusionExplanation** member of the classifier result indicates whether it was due to the confidence score being too low, or due to a failure of the classification validation check.

  Note: If the OCR Classifier tool has only one training character, the confidence score will be set to the score and the *confusionCharacter* will be default constructed.

- FAILED if the score does not satisfy the accept threshold.

  Note: If the highest scoring character does not satisfy the accept threshold, then the confidence score will be set to zero and the alternative character vector will be empty, and the confusion character will be a default constructed character.

Swap Characters

The OCR Classifier tool optionally takes a swap character set (**ccOCSwapCharSet** in *ch_cvl/ocswapch.h*) as input. If a swap character set is specified, then the confidence score is defined as the difference between the score of the highest scoring classification and the score of the next highest classification that is not swappable with the highest scoring classification.

Scale Filters

The OCR Classifier tool allows you to optionally specify x-scale/y-scale filter ranges, which specify the acceptable scale factors between the train-time character mark rectangles and the run-time character mark rectangles. Only classes/instances that satisfy the scale constraints are compared against the rectified image.

For diagnostic purposes, the OCR Classifier tool can optionally return which train classes/instances were not compared against the rectified image because they did not satisfy the scale constraints.

Handling Spaces

The OCR Classifier tool handles space characters differently from nonspace characters. In particular, you specify whether input characters are to be classified as spaces. The behavior of the tool is to pass through all space characters with the specified score, with no alternative characters, and with the confidence score set to be equal to the score.

Motivation for Multiple Training Instances

Providing the OCR Classifier tool with multiple training instances of each class allows the tool to better discriminate between similar class types. Providing multiple training instances has its advantage but also some disadvantages.

Advantage

Better classification performance can be expected, especially for discriminating between similar classes that exhibit intra-class variation. See the following figure for an illustration of similar classes with intra-class variation.

Similar classes with intra-class variation

Disadvantages

The following are the disadvantages of providing multiple training instances:

- Decrease in speed performance.
- Possible decrease in confidence score (because multiple training instances should not only improve the highest score but it should improve the second-highest score as well).
- Possible induction of overtraining; that is, the training of bad characters.

Note: You should only add multiple training instances when you are observing misreads, but in those cases, adding additional training instances can be an effective solution (that is, if the OCR Classifier tool is failing to classify or misclassifying characters, then new instances should be added).

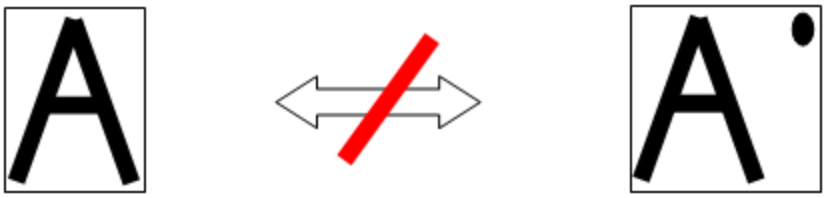
Recomputing Classification

The tool allows you to recompute the classification result for a character based on a 1D set of allowable character codes and for a line of characters based on a 2D set of allowable character codes.

What the OCR Classifier Tool Does Not Do

The following functionality limitations apply for the OCR Classifier tool (these functionalities are explicitly not included in the tool):

- The tool does not support *don't care* masks for the traintime image/region or for the runtime image/region.
- The tool does not match or classify runtime rectified images of characters that are of significantly different orientation than the traintime rectified instances.
- The tool does not handle polarity reversals between the traintime image and the runtime image (that is, if the training character was a white R against a black background, then the runtime character cannot be a black R against a white background). However, the tool can classify both dark-on-light characters and light-on-dark characters if it has been trained on examples of both polarities.
- The tool does not take a contrast threshold as input (such that it would pass classifications whose measured contrast exceeded that contrast threshold).
- The tool may fail to correctly classify a character whose rectified image's mark rectangle is substantially different from the bounds of the mark rectangle used for training. See the following figure for illustration.



Training image with mark rectangle versus runtime image with substantially different mark rectangle

OCR Dictionary Fielding Tool

The main purpose of the OCR Dictionary Fielding tool is to locate the read string in the specified fielding so that the read string can be reclassified based on the fielding.

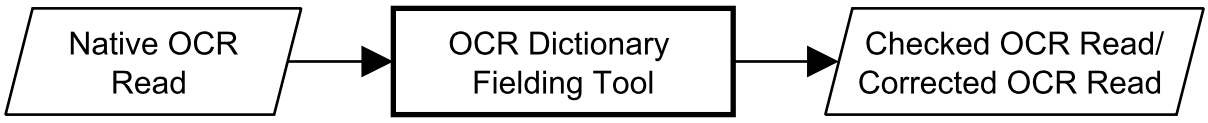
Also, The OCR Dictionary Fielding tool provides functionality to verify and correct strings read by the OCR Classifier tool taken as input. It returns the set of best matching valid strings.

There are two secondary usages:

- OCR result verification:
  In this usage, you try to determine whether the OCR result is correct or not by comparing it against acceptable results specified via fielding.

- OCR result correction:
  In this usage, the raw OCR result string is not among the acceptable results. You try to find an acceptable string allowed by the field that is closest to the raw OCR result string.

The following figure shows these secondary usages of the OCR Dictionary Fielding tool.



OCR Dictionary Fielding tool typical usage

Note: Each character in a string is characterized by a 32-bit value, but usually 21 bits (UNICODE) suffice.

A typical situation where fielding is used is when the input data includes excess prefix/suffix characters and fielding should ignore those prefix/suffix characters. In this case, the OCR Dictionary Fielding tool determines the offset of the string position in the input data.

The OCR Dictionary Fielding tool takes as input the minimum and maximum string length of the result strings, which define a set of possible fieldings (subsets of the given fielding vector). The fielding algorithm tries each of these possible fieldings, and for each fielding, finds the best matching position in the string (the position which induces the most matched characters).

Note that for the fixed-length fielding mode, you may always ignore prefix/suffix characters. For variable-length fielding, if the maximum length is specified, then the OCR Dictionary Fielding tool is allowed to trim as many characters as necessary to satisfy the string length requirement. For variable-length fielding, the tool will optionally trim failing leading/trailing characters; you can turn on or off this capability.

The OCR Dictionary Fielding tool includes the ability to detect missing fielded characters at the beginning or end of a string. This detection of missing fielded characters is accomplished by allowing the fielding to be positioned “off” the runtime character input string. That means that if the input data only included n characters and the fielding only included n characters, then there are actually 2 x n+1 candidate positions for the fielding. Of course, the fielding result prefers fielding strings that overlap the run-time string as much as possible.

The OCR Dictionary Fielding tool supports timeouts.

Note:
There are no constraints on string length (for the dictionary, input string, fielding, or the regular expression).

There are no constraints on the size of the dictionary.

Swap Characters

The OCR Dictionary Fielding tool allows you to specify swap characters, which are characters that should be considered equivalent as allowed by a particular application (usually because the characters are so similar that it is acceptable for the application to treat them as interchangeable). The tool can freely substitute swap characters without incurring any penalty. For example, if the input string was “5UPPLY”, the fielding allowed only uppercase letters A through Z for the first character position, and the swap character set included “5” == “S”, then the tool would return “SUPPLY” as the valid result string.

Note:
The OCR Dictionary Fielding tool favors the original character over the swapped characters, in that the tool favors strings that match the original characters over strings that match the swapped characters of the original characters.

The swap characters are 1-to-1 (such as “5” for “S”). Many-to-1 swapping (such as “m” for “rn”) is not supported.

For more details, see the *ch_cvl/ocswapch.h* header file.

Multiple Characters at Each Position

In addition to supporting swap characters, the OCR Dictionary Fielding tool also allows for the possibility of multiple candidates for each character in the input string. Multiple candidates can occur at run-time if there are candidates with scores almost as high as the score of the highest scoring candidate. For example, a damaged character may resemble both a “6” and a “9”. In such situations, it might be inappropriate to simply select one of the possibilities as the true correct answer and treat similar possibilities as completely incorrect. It is better to consider all of them, while slightly favoring the highest scoring candidate.

The OCR Dictionary Fielding tool handles these situations by allowing you to specify one or more candidates at each character position. There is always one best primary candidate and some secondary candidates (the latter referred to as alternative classes/ characters for the OCR Classifier tool). If none of the classification scores satisfied the *acceptThreshold*, then the best primary candidate will be *eUnknown*.

A result string that uses the primary candidate is considered better than one that uses a secondary candidate.

Handling Unfielded Spaces

The OCR Dictionary Fielding Tool might get an input string that includes extraneous/superfluous space characters. You can specify whether the tool should automatically skip unfielded input spaces.

Applied Rules During Fielding

The following table shows the rules that are applied during fielding in the following order of preference in case **isGoodMetric()** == *usePrimary*.

Highest level character matching fielding	Score Contribution	Character Result	Status
Primary character	Score	Primary character	PASS Primary character
Swappable character of primary character	Primary character's score	Swappable character of primary character	PASS Primary swap character
Secondary character	0	Secondary character	FAIL Secondary character
Swappable character of secondary character	0	Swappable character of secondary character	FAIL Secondary swap character
No match - but fielding only includes a single character	0	Fielding character	FAIL Substituted character
No match but fielding contains a space	0	Space	FAIL Substituted Space
No matching character or unknown character	0	No character	FAIL No match
No character provided	0	No character	FAIL No character

Applied rules during fielding

The following table shows the rules that are applied during fielding in the following order of preference in case **isGoodMetric()** == *isPrimaryOrSecondary*. The score of secondary characters/secondary swaps is not clamped to 0.

Highest level character matching fielding	Score Contribution	Character Result	Status
Primary character	Score	Primary character	PASS Primary character
Swappable character of primary character	Primary character's score	Swappable character of primary character	PASS Primary swap character
Secondary character	Score	Secondary character	PASS Secondary character
Swappable character of secondary character	Score	Swappable character of secondary character	PASS Secondary swap character
No match - but fielding only includes a single character	0	Fielding character	FAIL Substituted character

Highest level character matching fielding	Score Contribution	Character Result	Status
No match but fielding contains a space	0	Space	FAIL Substituted space
No matching character or unknown character	0	No character	FAIL No match
No character provided	0	No character	FAIL Inserted character

Note:
The OCR Dictionary Fielding tool only returns a single result.
The OCR Dictionary Fielding tool only supports a single fielding set for a dictionary.

What the OCR Dictionary Fielding Tool Does Not Do

The OCR Dictionary Fielding tool does not take as input the image location of the characters, and is therefore not influenced by the character positions. In particular, the tool will not try to match SPACEs based on character image locations.

Reclassifying after Fielding

The **cfOCRReclassifyAfterFielding()** global function contained in the *ch_cvl/ocrglue.h* header file allows you to reclassify after fielding by recomputing the OCR Classifier tool's result based on the OCR Dictionary Fielding tool's result. For more information, see the *CVL Class Reference*.

Using the OCR Tool

This section describes how to use the OCR tool.

The following OCR-specific header files are available for the following OCR components.

Tool	Header file
OCR Segmenter tool	*ch_cvl/ocsegmnt.h*
	ch_cvl/occhar.h
OCR Classifier tool	*ch_cvl/ocrclass.h*
	ch_cvl/occhar.h
	ch_cvl/ocswapch.h
OCR Dictionary Fielding tool	*ch_cvl/ocrfldng.h*
	ch_cvl/ocrdcbas.h
	ch_cvl/ocswapch.h
	ch_cvl/ocrddefs.h
All	*ch_cvl/ocrdefs.h*
	ch_cvl/charcode.h

The *ch_cvl/ocfont.h* header file contains the **ccOCFont** class, which represents a set of characters (including their character codes, images, and other metadata) primarily for use with OCR and OCV and also more generally to represent a font.

The *ch_cvl/ocrglue.h* header file contains the **cfOCRReclassifyAfterFielding()** global function, which allows you to recompute the classifier result according to the fielding result.

The *sample/cvl/ocr.cpp* file is a single-file sample code that demonstrates the use of the OCR components.

OCR Segmenter Tool Input and Output Information

The *ch_cvl\ocsegmnt.h* header file provides the functionalities to provide the input parameters to and obtain the output parameters and results from the OCR Segmenter tool.

Input Information

The OCR Segmenter tool takes the main inputs listed in the following table.

Input	Description
Input image	8-bit greyscale input image, provided as an input parameter for the **cfOCSegmentCharacters()** global function.
Search region	The ROI, provided as an input parameter for the **cfOCSegmentCharacters()** global function. ROI is specified as an affine rectangle in the client coordinates of the image, containing at most one string and some surrounding background (but no other significant features, such as part of a different nearby line of characters). The x-axis should be approximately parallel to the baseline of the line of characters; the y-axis indicates the nominal skew, if any. The affine rectangle must be entirely contained in the image.

OCR Segmenter tool inputs

You can supply the following further input information to the OCR Segmenter tool:

- Analysis mode (minimal or standard)

For both the standard and minimal modes:

- Character polarity
- Angle and skew search settings
- Normalization mode
- Stroke width filtering
- Foreground threshold level
- Character fragment handling
- Size requirements for the characters to be reported
- Intra- and inter-character gap handling
- Wide character splitting settings
- Character minimum aspect setting
- Expected character width variation in a font
- Space character handling

For the standard analysis mode only:

- Pitch settings

You supply this information to the OCR Segmenter tool as a set of input parameters for the **cfOCSegmentCharacters()** global function.

Note that OCR Segmenter tool is capable of handling essentially arbitrary scale range (in terms of a classifier's font, although the tool has no knowledge of any font, which means it has no concept of scale in itself either, only of character size). However, for some cases, you will need to specify parameters such as the minimum or maximum character width, minimum pitch, and so on; and those are most powerful when they can be set tightly, which ideally means looking for characters at a known scale.

The OCR Segmenter tool can optionally insert space characters into the gaps between other (non-space) characters. You can specify whether the tool should:

- Never insert spaces.
- Insert at most one space character per gap.
- Insert zero or more space characters based on the width of the gap.

You can also specify how space characters are scored:

- Space characters always get a score of 1.0.
- Space characters get a score based on the fraction of pixels that are background.

You can specify a minimum and a maximum width of space characters to allow the OCR Segmenter tool to determine the appropriate number of space characters (possibly zero) to insert into a gap.

You use the **reset()** function to reset all parameters to default values.

Output Information

Some of the output parameters you obtain from the OCR Segmenter tool are the following:

- Rectified image of the line.
- Whether this object contains an image of the line of the segmented characters after rectification and normalization.
- Rectified and normalized image of the line.
- Binarization threshold and whether inverted thresholding is needed for the normalized image to form the binarized image.
- Enclosing affine rectangle of all the segmented characters' mark rectangles.
- Set of position results, one for each segmented character. See the **ccOCCharSegmentPositionResult** class.

For the complete list, see the **ccOCCharSegmentPositionResult**, **ccOCCharSegmentLineResult**, **ccOCCharSegmentParagraphResult**, **ccOCCharSegmentResult** classes in the *CVL Class Reference*.

Note that the rectified and normalized image of the line, binarization threshold, and whether inverted thresholding is needed are intended primarily to provide diagnostic feedback.

The **ccOCCharSegmentPositionResult** class contains the following OCR Segmenter tool position results:

- Whether this position result contains a character and possibly other computed information.
- The segmented character.
- The enclosing rectangle of the foreground pixels in the character.
- The enclosing rectangle of the character, including both foreground pixels and typically some surrounding background pixels.
- Whether the result character is a space.
- Score of the result character if it is a space.

You use the **reset()** function to reset the result to a default state.

OCR Classifier Tool

The *ch_cvl\ocrclass.h* header file provides the functionalities for the OCR Classifier tool.

Training the OCR Classifier Tool

One possible way you can train a font is by presenting some images to the system, specifying affine rectangles around each image's string and the actual strings. The OCR Segmenter tool automatically segments each image and provides

an affine rectangle around each character.

Note: You do not have to provide images of strings; you can also provide images of single characters, in which case training involves more than one image.

You verify that each affine rectangle surrounds the appropriate character, and then tell the OCR Classifier tool to train all the characters. The OCR Segmenter tool generates rectified images for each of the affine rectangles. As part of the training procedure, the OCR Classifier tool accepts these rectified images (one for each character) and the associated character code as training input. You continue training characters in this manner until you decide that they have trained enough characters.

Running the OCR Classifier Tool

At runtime, the OCR Segmenter tool segments a subwindow and determines affine rectangles around each candidate character. The OCR Segmenter tool generates rectified images for each affine rectangle. As part of the runtime procedure, the OCR Classifier tool accepts these rectified images as input and reports the determined character code(s) and respective scores. For each rectified image, the OCR Classifier tool returns the highest scoring class, its score, as well as one or more secondary classes (and their respective scores).

An Alternative or Complementary Training Strategy

An alternative or complementary training strategy is to free-run the machine (automatically presenting images and running OCR) and asking the user to identify characters for which the OCR Classifier tool failed to classify. This strategy could be termed Train-on-Failure. Note that this strategy would probably only succeed if the OCR Classifier tool were biased towards returning FAILED rather than the wrong class.

OCR Classifier Tool Classes

The OCR Classifier functionality involves the following classes.

Class	Description
ccOCRClassifierTrainParams	Training parameters for the OCR Classifier tool, see the *ch_cvl/ocrclass.h* header file.
ccOCRClassifierRunParams	Run-time parameters for the OCR Classifier tool, see the *ch_cvl/ocrclass.h* header file.
ccOCRClassifierPositionResult	All result data for the classification of a run-time character, see the *ch_cvl/ocrclass.h* header file.
ccOCRClassifierLineResult	All result data for the classification of one line of characters, see the *ch_cvl/ocrclass.h* header file.
ccOCRClassifier	The OCR Classifier tool, see the *ch_cvl/ocrclass.h* header file.
ccOCSwapCharSet	A set of swap character groups. Characters within each group are swappable with each other. See the *ch_cvl/ocswapch.h* header file.

For detailed information, see the *CVL Class Reference* and the *ch_cvl/ocrclass.h* header file.

OCR Classifier Tool Training Inputs

You can supply the following training inputs to the OCR Classifier tool.

Input	Description
Training characters	The characters to be trained.
Training parameters	The training parameters.

OCR Classifier Tool Run-Time Inputs

You can supply the following run-time inputs to the OCR Classifier tool.

Input	Description
Run-time character(s)	The input character image(s).
Swap character set	The set of swap character codes.

Input	Description
Is space	Whether this character is a space.
Space score	Space score (if this character is a space).
Run-time parameters	The run-time parameters.

OCR Classifier Tool Training Parameters

You specify the following training parameters for the OCR Classifier tool.

Parameter	Description
Template size	The size to which the rectified images are resampled.
Maintain aspect ratio	Whether to maintain the rectified images' aspect ratio when resampling the rectified image at train-time/run-time.
Algorithm	The algorithm used for classification.
Image preprocessing	The image preprocessing methods used for classification.

Note:
The OCR Classifier tool includes functionality for displaying the subsampled templates.

There is no training step for a font; a font is just a collection of bitmaps and ancillary information. The OCR Classifier tool trains from the rectified images. To create a new font, grab it from the OCR Segmenter tool and save it directly.

OCR Classifier Tool Run-Time Parameters

You specify the following run-time parameters for the OCR Classifier tool.

Parameter	Description
Accept threshold	The minimum score in order for a match to be accepted.
Confidence threshold	The minimum score differential (between the highest scoring class and the next-highest and non-swapped scoring class) for a match not to be confused.
Use x-scale filter	Whether to use the x-scale filter.
X-scale filter	X-scale filter range for skipping candidate classes/instances whose rectified training image's x-size (that is, width) is beyond the range specified here.
Use y-scale filter	Whether to use the y-scale filter.
Y-scale filter	The y-scale filter range for skipping candidate classes/instances whose rectified training image's y-size (that is, height) is beyond the range specified here.
Report skipped train character indices	Whether to include in the result object the set of train character indices that were skipped because they did not satisfy the scale constraints.
Keep processed image	Whether to keep around the processed image corresponding to the run-time input rectified image.

Classification Results for a Run-Time Character

The following classification results for a run-time character are available provided by the OCR Classifier tool:

- Classification
- Status information (READ, CONFUSED, or FAILED)
- Confusion explanation – a reason that leads to the CONFUSED status.
- Confusion character
- Confidence score
- Alternative characters
- Skipped training character indices
- Processed image

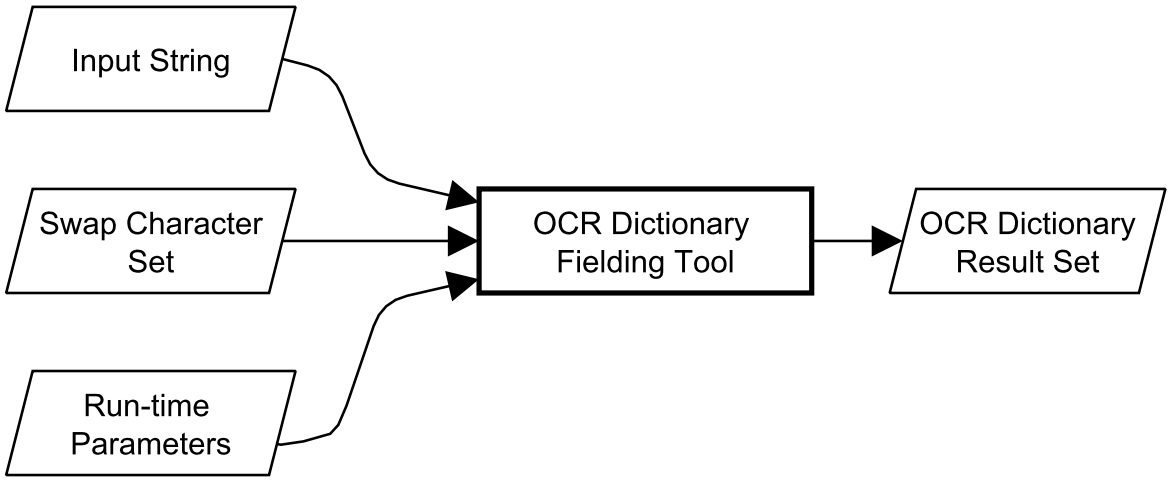
Note: These classification results are also available for one line of characters through the **ccOCRClassifierLineResult** class.

Thread Safety

The OCR Classifier tool follows the CVL convention of thread safety. CVL functions can be called on different objects in different threads, but different threads cannot concurrently call functions on the same CVL objects.

OCR Dictionary Fielding Tool

The following figure illustrates how OCR Dictionary Fielding tool classes work together.



How OCR Dictionary Fielding classes work together

Using the OCR Dictionary Fielding tool involves the following steps:

1. Set up the fielding to characterize acceptable result strings (equivalent to training).
2. Set up the parameters and specify the swap characters.
3. Convert OCR Classifier tool results into an input string to the OCR Dictionary Fielding tool.

   Note: In a typical usage, each character position in the OCR Dictionary Fielding tool's input string would include the following:

   a. If the OCR Classifier tool's position result status is *eRead* or *eConfused*, both the primary character and all alternative characters with score greater than or equal to the accept threshold in the OCR Classifier tool's position result.

   b. If the OCR Classifier tool's position result status is *eFailed*, only one value: *ccCharCode::eUnknown*.

4. Run the OCR Dictionary Fielding tool on the input string and examine the results.

OCR Dictionary Fielding Tool Classes

The OCR Dictionary Fielding tool uses the following classes.

Class	Description
ccOCRDictionaryChar	Characters used in the strings, with a character code and a score. See the *ch_cvl/ocrdcbas.h* header file.
ccOCRDictionaryString	Strings of characters. See the *ch_cvl/ocrdcbas.h* header file.

Class	Description
ccOCRDictionaryCharMulti	A collection of character candidates for each character position, with a primary candidate and potentially multiple secondary candidates. See the *ch_cvl/ocrdcbas.h* header file.
ccOCRDictionaryStringMulti	Strings of multicharacters, where each position has a **ccOCRDictionaryCharMulti** object. This is the most general form of string representation for dictionary input and result strings. See the *ch_cvl/ocrdcbas.h* header file.
ccOCSwapCharSet	Characterizes a set of swap characters. See the *ch_cvl/ocswapch.h* header file.
ccOCRDictionaryFielding	This is the OCR Dictionary Fielding tool. It characterizes a set of acceptable character choices for each character position in a string. See the *ch_cvl/ocrfldng.h* header file.
ccOCRDictionaryFieldingRunParams	A set of run-time parameters to specify optional behaviors of the OCR Dictionary Fielding tool. See the *ch_cvl/ocrfldng.h* header file.
ccOCRDictionaryResult	Contains detailed results about the input string, the result string, their quality ratings, and so on. See the *ch_cvl/ocrdcbas.h* header file.
ccOCRDictionaryResultSet	Contains all results in checking a run-time input string. See the *ch_cvl/ocrdcbas.h* header file.

Input String

If a segmented character is unclassified, it is input as an unclassified character, not an empty vector. Each element in the input string is a vector which corresponds to the possible characters at that position; each such element must contain at least one entry.

Fielding Vector

You use the **positionFieldings()** function of the **ccOCRDictionaryFielding** class to specify the fielding vector.

The fielding vector can be either a vector of valid characters or a combination of OR-able flags. The following table summarizes these OR-able flags.

Flag	Character type
eAlphaUppercase	A - Z
eAlphaLowercase	a - z
eNumeric	0 - 9
eSpace	The space character (UTF-32 character code 0x20).
eAnyNonspaceCharacter	Any character not in the Cognex-reserved UTF-32 code ranges except the space character whose UTF-32 character code is 0x20.
eAnyCharacter	Any character not in the Cognex-reserved UTF-32 code ranges.
eAlpha	A - Z and a - z
eAlphaNumeric	A - Z, a-z, and 0 - 9
eAlphaUppercaseNumeric	A - Z and 0 - 9
eAlphaLowercaseNumeric	a - z and 0 - 9
kDefaultFielding	Any non-Cognex-reserved nonspace character.

Note:
The enum type **ccOCRDictionaryDefs::Fielding** provides values to represent some pre-defined sets of characters. Among these are *eSpace* and *eAnyNonspaceCharacter*.

eSpace explicitly refers to the UTF-32 character code 0x20 and does not include any other character codes that are otherwise considered as whitespaces by the UTF-32 standard.

eAnyNonspaceCharacter includes all UTF-32 character codes except 0x20 and those in the Cognex-reserved code ranges (see the *ch_cvl/charcode.h* header file).

Empty Fielding Vector Handling

If the fielding vector is empty (**numPositions()**==0), then the input string is unconditionally accepted. In other words, an empty fielding vector is treated as an infinitely long wildcard fielding. Every character of the *inputString* is considered *ePrimary*, but *inputString* characters of *ccCharCode::eUnknown* are considered *eNoMatch*.

Swap Characters

You are not required to create multi-swap characters to specify the swap characters; you only need to specify sets of equivalent characters.

OCR Dictionary Fielding Tool Run-Time Parameters

You can control the OCR Dictionary Fielding tool's behavior by using the following parameters:

Parameter	Description
Minimum string length	The minimum length of the result strings.
Maximum string length	The maximum length of the result strings.
Ignore unfielded spaces	Allows the tool to ignore/skip spaces in the input string where the corresponding fielding settings for those character positions do not allow spaces.
Fixed length fielding	Whether to run in the fixed-length mode or the variable-length mode. The fixed-length mode uses the specified fielding vector in its entirety, while the variable-length mode allows using a subsequence.
Maximum fielding first index	For variable-length fielding, the fielding subsequences to be considered must start at a position no greater than this index.
Minimum fielding last index	For variable-length fielding, the fielding subsequences to be considered must end at a position no smaller than this index.
Ignore failing prefix and suffix	For variable-length fielding, allows the tool to discard leading and trailing failed-to-read characters without incurring penalty.
Is good metric	Specifies whether to treat only primary character matches as good or to treat either primary or secondary character matches as good.

Note: Results are guaranteed to have strings with lengths between the minimum string length and the maximum string length.

OCR Dictionary Fielding Tool Outputs

The OCR Dictionary Fielding tool generates a result set containing a single result object. The result object contains detailed information for each output string on how the tool checked the input string and created the result string. Besides the result string itself, the **ccOCRDictionaryResult** class also contains the following details:

- The start position (as the character index) of the result string in the input string.
- The number of ignored prefix and suffix characters in the input string.
- The number of primary and secondary characters and optionally their swaps in the result string.
- An overall flag indicating whether this result passed dictionary checking.
- An overall status for the input string to indicate its quality rating as well as the statuses of individual characters in the input string.
- An overall status for the result string to indicate its quality rating as well as the statuses of individual characters in the result string.

The input and result strings are assigned an overall quality rating that is the lowest quality rating among all of its characters. The quality ratings are shown in the following table in descending order.

Status	Applies to
ePrimary	Both input and result strings and characters
ePrimarySwap	Both input and result strings and characters
eDeletedIgnored	Only input string and characters

Status	Applies to
eSecondary	Both input and result strings and characters
eSecondarySwap	Both input and result strings and characters
eSubstituted	Both input and result strings and characters
eSubstitutedSpace	Both input and result strings and characters
eInserted	Only result string and characters
eDeleted	Only input string and characters
eNoMatch	Both input and result strings and characters

eSubstituted is ranked higher (in terms of quality) than *eSubstitutedSpace*. The reason behind this is that *eSubstituted* occurs when the fielding contains a unique match character, whereas *eSubstitutedSpace* occurs when the fielding contains a space character (in addition to other allowable characters).

Threading

The OCR Dictionary Fielding tool satisfies the CVL object-single-threaded requirement. The tool may be multithreaded (that is, enabling multiple threads might improve performance).

Fielding Example

The following is a simple example for using fielding to check OCR strings of the form “MAR 2011”.

```
ccOCRDictionaryFielding fielding;

{

cmStd vector<ccOCRDictionaryPositionFielding> positionFieldings;

positionFieldings.push_back(ccOCRDictionaryDefs::eAlphaUppercase);

positionFieldings.push_back(ccOCRDictionaryDefs::eAlphaUppercase);

positionFieldings.push_back(ccOCRDictionaryDefs::eAlphaUppercase);

positionFieldings.push_back(ccOCRDictionaryDefs::eSpace);

positionFieldings.push_back(ccOCRDictionaryDefs::eNumeric);

positionFieldings.push_back(ccOCRDictionaryDefs::eNumeric);

positionFieldings.push_back(ccOCRDictionaryDefs::eNumeric);

positionFieldings.push_back(ccOCRDictionaryDefs::eNumeric);

fielding.positionFieldings(positionFieldings);

}

ccOCRDictionaryString inputString(_T("abcMAR 2011xyz"), 1.0);

ccOCRDictionaryStringMulti inputStringMulti(inputString);

ccOCSwapCharSet swapCharacterSet;

ccOCRDictionaryFieldingRunParams fieldingRunParams;

ccOCRDictionaryResultSet resultSet;

fielding.run(inputStringMulti, swapCharacterSet, fieldingRunParams,

resultSet);
```

Speed

Tool execution times are longer with longer fonts, including cases where there are multiple instances of each character; specifically, a font with more instances of each character may run more slowly than a font with fewer instances of each

character.

OCR Segmenter Tool Speed

The execution time of the OCR Segmenter tool depends more on the area of the ROI than on the number of characters, which typically means it also depends on the size of the characters (because larger characters usually mean a larger ROI). Also, the OCR Segmenter tool time does not depend on the number of characters in the font, which means that because classifier time increases with the number of characters in the font, the OCR Segmenter tool time should become a smaller fraction of the overall time as the font gets larger.

OCR Classifier Tool Speed

The OCR Classifier tool's speed may increase linearly with font size and character area.

OCR Font

The OCR Font class (**ccOCFont**) is a container class for a font, representing a set of characters, including their character codes, images, and other metadata, primarily for use with OCR and OCV and also more generally to represent a font.

The **ccOCFont** class supports serialization to and from **ccArchive**. The class provides the **ccOCFont::load()** and **ccOCFont::save()** functions to load and save the file using the file name as the sole argument.

The font file may be in ".ocr" format or the older ".ocf" or ".ocm" formats.

Although **ccOCFont::load()** can load files in the older formats, trying to train the OCR Classifier tool using such a font (produced either manually or by a different Cognex tool such as the Image Font Extractor) and then running the OCR Classifier tool on images produced by the OCR Segmenter tool will generally give a worse OCR read rate (for example, percentage of correctly classified characters) than creating a font using the OCR Segmenter tool.

OCV Tool

This chapter describes the OCV tool, a vision tool for performing optical character verification. It contains the following sections:

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

acceptance threshold: If the **matching score** is below this level the character at the specified position fails verification.

character line: A spatial arrangement of characters you expect to find in your application. For example, a character string. Sometimes referred to as a line.

character position key set: A set of keys that define allowed characters for a single character position in a **character line**.

confidence threshold: If the difference between the **matching score** for the position and the highest scoring confusion character is greater than the confidence threshold, the position passes verification; otherwise, the verification status of the position is confused.

confusion score: A number in the range 0 through 1 that indicates the probability that two characters can be confused with one another during optical verification. A value of 0 means that the characters will never be confused with each other; and a value of 1 means that they will always be confused.

confusion threshold: If the computed confusion score for a character pairing is greater than the confusion threshold, the characters are easily confused with each other during optical verification.

current keys: A set of keys selected from the **character position key set** that are used with the current run-time image.

character line arrangement: A sequence of **character lines** in space. Sometimes referred to as a line arrangement.

matching score: The score generated when the OCV tool measures the match between the run-time character image for a position and the model image for a character expected to be in that position. A score is calculated for each of the **current keys** and the highest score is the matching score.

The OCV tool also reports score results for **character lines** and **character line arrangements**.

optical character verification (OCV): The process of verifying that the characters in a text area are the expected characters.

OCV Tool Overview

The OCV tool performs *optical character verification* (OCV), a process for verifying that each character in a text area is the expected character in the expected location. The OCV tool tolerates variations in

- lighting
- image quality
- font stroke width

- character position, rotation, and scale
- line position and rotation
- line arrangement position, rotation, and scale

The OCV tool's primary application is to verify date and lot codes printed on product labels, but you can use it with any vision application that requires verifying letters, numbers, or symbols.

How the OCV Tool Works

When you run the OCV tool, it does the following:

- It locates a target line arrangement based on a fixture and fixture offset.
- It measures the match between the run-time image where the character is expected and the model image for the expected character.
- It evaluates the verification status for the character based on the position's accept threshold and confidence threshold.
- It verifies the line arrangement if all lines in the arrangement pass verification. A line fails if any of the character positions fails verification.
- It reports whether it successfully verified the entire text area, and it returns information about the verification status and run-time coordinate positions of characters, lines, and line arrangement.

Training the OCV Tool

When you train the OCV tool, you first create models for each character you want to verify. You then assemble an alphabet of these character models. The alphabet defines the set of character models from which you can define lines of characters. You then construct a line arrangement from the lines of characters. Finally, you create an OCV tool, supplying the defined line arrangement, and call the tool's **train()** function.

Creating a Character Model

A character model represents a single character to verify. You can create two types of character models, normal or blank.

A normal character can be verified by matching its shape. For example, any alphanumeric character or distinctive shape such as a logo, a group of letters, or a symbol is a normal character. The model for a normal character includes a bound image of the character.

A blank character represents a "white space" character such as a space or tab, and can be verified by the absence of shape. The model for a blank character includes an unbound image of the character, which provides information necessary to obtain a client to image transformation.

The OCV tool verifies blank characters after verifying all other characters. It uses information extracted from neighboring normal characters to properly verify the blank character.

The client coordinate system of the bound or unbound image defines the character model's coordinate system. The model area is determined by a rectangle defined in image coordinates.

You can obtain images for character models by loading them from individual records of a Cognex image database (CDB) file. Alternatively, you can load an image that contains many characters and then select the area of each character in image coordinates as you construct the model for that character.

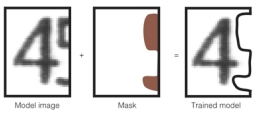
You refer to a specific character model using its *key*. Typically, you would use the ASCII code for the character as the key for its model. A character model also has a *name* string with which you can refer to multiple instances of the same character and a string used to *describe* the model.

Character Masks

Both normal and blank characters can have associated masks. A mask is an image that denotes certain areas of the character image that the OCV tool should ignore when training.

If you specify a mask image, only those pixels with a value greater than or equal to 128 are included in the character model. Pixels with a value less than 128 are not included. An unbound mask is equivalent to a mask image filled with “care” pixels. The OCV tool ignores the client coordinate transform of the mask image.

The figure below shows an example of how you might use a mask image to exclude information from the model image when you train a model.

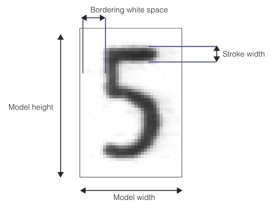


Using a mask image to train a character model

When the model is applied at run-time, the excluded area in the trained model has no effect on the score of the character.

Character Model Image Guidelines

The figure below shows an example of a character model. For best results, you should observe the guidelines listed in this section when you select an image to create a character model.

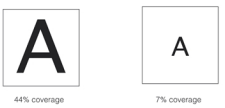


Character model.

- The image should be representative of a typical run-time character.
- The size of the character model image should be at least 10x10 pixels. Character model images larger than 64x64 pixels may cause the tool to operate more slowly.
- The stroke width of the character in the image should not vary by more than 10%.
- No more than 10% of the character should be missing. If you are attempting to create a font of dot-matrix characters, you should use an image processing tool to fill in the character stroke for both character model and run-time images.
- The character model image should contain a border of white space around the character that is between one and three times the stroke width.

Coverage

When you create a non-blank character model, you provide a bound image and a rectangle (**ccOCModel::imageArea**). The ratio of the supplied image size to the size of the rectangle that is occupied by the character being trained has an effect on the score produced when you verify the characters. The figure below shows how the proportion o f the rectangle occupied by the character can vary.



Model and image size

When the OCV tool locates and scores characters, it computes the score based on the total image area that you specify when you create the character model. You should make sure that all of the character models that you create have approximately the same coverage ratio. If the coverage ratios differ widely, the tool may not score characters consistently. Also, make sure to observe the guidelines for stroke width and white space border size listed in the section *Character Model Image Guidelines* on page 498.

Creating an Alphabet

An alphabet is a set of characters from which you define lines of text to verify. Typically, the alphabet comprises characters from a single font. You supply a unique key for each character in the alphabet—for example, the ASCII code for the character.

The alphabet incorporates a table of confusion scores calculated for pairings of character models in the alphabet. Each confusion score in the table indicates the probability, on a scale of 0 to 1, that the compared pair of characters will be confused with each other during optical verification. The confusion scores are based on comparing the characters at the same position, angle, and scale. The table below shows the confusion score table for an alphabet that has the letters A, B, and C and the numbers 7, 8, 9.:

	A	B	C	7	8	9
A	1.0	.75	.50	.65	.70	.60
B	.75	1.0	.70	.55	.90	.80
C	.50	.70	1.0	.50	.65	.50
7	.65	.55	.50	1.0	.60	.65
8	.70	.90	.65	.60	1.0	.85
9	.60	.80	.50	.65	.85	1.0

Confusion score table for characters of an alphabet

The greater the confusion score for a character pair, the greater the similarity between the characters. Comparing a pair of the same character yields a confusion score close to or equal to 1.0.

The confusion score table may not always contain symmetric values. That is, the confusion score calculated for A compared with B may be different than the confusion score obtained for B compared with A.

Confusion Threshold

You must also set the *confusion threshold* value for the alphabet. The confusion threshold determines whether two characters can potentially be confused, based on their confusion score. If the confusion score for a pair of characters is greater than the confusion threshold, the OCV tool may be confused by the similarities between the two characters during optical verification.

If you set a lower confusion threshold, the OCV tool compares more characters with the expected character. This yields better verification results, however it also requires more time for the tool to run. You can inspect the confusion score table to facilitate choosing an appropriate confusion threshold.

Confusion Override Keys

By default, as described above, the OCV tool automatically determines which characters are potentially confusing by considering the confusion matrix and confusion threshold. The OCV tool lets you override this method of determining which characters are confusable by specifying an explicit list of *confusion override keys*.

If you specify confusion override keys for a particular character position within a particular line, the OCV tool will use the characters associated with the override keys as the confusable characters. You must specify override keys for each character position individually

Specifying override keys lets you perform OCR (Optical Character Recognition) using the OCV tool. By specifying which characters are confusable, you can force the tool to check for confusion among the characters defined as part of the current key set. This lets you determine not only *that* one of the valid characters is present, but *which* one is present.

Creating a Line of Characters

You create a line object for each line of text your application should verify. A line object represents a spatial arrangement of characters from one alphabet. Each line in a group of lines may use characters from a different alphabet. A typical line consists of evenly spaced, upright characters. However, you can also define a line made up of an arbitrary arrangement of characters.

To create a line:

1. Define a set of character keys for each line position.
2. Specify the spacing or poses and the uniform scale and uncertainties for the characters.

The OCV tool verifies blank characters after verifying all other characters. It uses information extracted from neighboring normal characters to properly verify the blank character. The tool identifies neighboring characters in the order you have specified the characters in the line, which may not always be sequential. For optimal verification results, the character before and after a blank character in line order should be a normal character that is physically closest to the blank character.

Defining Key Sets for Character Positions

For each character position in a line, you supply a set of keys for the characters that may appear in that location. You also select the indexes of the keys that are to be the current key for the character position. A position can have any number of current keys from the set of character keys. The OCV tool uses the current keys to determine the characters to compare with the run-time character image found, allowing you to verify one of multiple characters at a given line position.

If you include a current key index of -1 in the current key set for a character position, anything found at the position will pass verification. This is the same as setting a wildcard for the character position. A wildcard takes precedence over all other characters in the key set.

Specifying Confusion Override Keys

As described in the section *Confusion Override Keys* on page 499, you may specify a specific set of keys to use as confusion override keys for a key set. If you specify confusion override keys, then the tool will use only these keys as the confusable characters for the position.

Changing the Current Key

Allowing a set of keys for characters that may possibly appear at a position makes the OCV tool more efficient and useful at run-time. You can switch a current key or keys between runs of the tool by calling the **cfOCChangeCurrentKey()** or **cfOCChangeCurrentKeys()** functions without needing to call the time-consuming **train()** function.

The new key may be any of the other keys within the key set for that position or a wildcard. Note that after calling either function to change current keys, you must call the tool's **retrain()** function. For more information about the **retrain()** function, see *Retraining the Tool* on page 502.

Character Spacing, Pose, and Uncertainty

A line object also represents information about the spacing, pose, and uncertainty in position for each character. You can specify one of the following:

- Uniform character spacing, pose, and uncertainty for all characters

  All characters are uniformly spaced with respect to their origins. The supplied scaling factor determines the scale of all characters. All characters have the same uncertainties in translation, angle, and scale. To specify the typical case of a horizontal line of upright characters (x- and y-axes of all characters parallel with the x- and y-axes of the first character in the line), specify a spacing that has only an x-axis translation component. This first method works well with nonproportional fonts.

- Uniform uncertainties for all characters, in addition to individual poses for each character

You arrange each character according to an arbitrary rigid pose (position and angle) with respect to an implicit line coordinate system. The scaling factor you supply determines the scale of all characters. All characters have the same uncertainties in translation, angle, and scale.

- Individual pose and uncertainty factors for each character

  You can supply arbitrary pose and uncertainty factors for each character. Characters are arranged according to rigid poses with respect to the implicit line coordinate system, and each character is scaled by its own scaling factor. You can also specify the uncertainty in translation, rotation, and uniform scale for each character in the line. This method applies well to proportional fonts.

The table below summarizes the various methods for specifying the spacing, pose, and uncertainty of characters in a line of text.

Method	Spacing	Pose	Uncertainty
1	uniform		uniform
2		arbitrary	uniform
3		arbitrary	arbitrary

Methods of specifying spacing, pose, and uncertainty for each character

Note:
You should not attempt to specify a scale uncertainty value larger than 0.05 (which specifies a range of expected scale from 95% to 105% of the trained size).

The mostly likely reason to need to specify a larger uncertainty value is to compensate for inaccurate calibration information stored in the image's client coordinates. The best approach to solving this problem is to re-calibrate the system and use a more accurate set of client coordinates.

Creating Line Arrangements

A line arrangement represents a sequence of lines in space. Lines are located with respect to the implicit coordinate space of the line arrangement. A typical line arrangement consists of parallel, horizontal lines of upright characters with uniform space between each line. However, you can also configure an arbitrary arrangement of lines.

To create a line arrangement, you decide which lines you have created are part of the line arrangement, and you specify the order of these lines. You also indicate the spacing or pose, in addition to the uncertainty of the lines in the arrangement.

Note: A line arrangement is not valid if any line in the arrangement contains blank characters for all character positions, regardless if other characters are present as part of the current key set for the character position.

Line Spacing, Pose, and Uncertainty

You can construct a line arrangement with the following types of spacing, pose, and uncertainty factors:

- Uniform spacing and uncertainty for all lines

  You construct a line arrangement that has lines spaced uniformly with respect to their origins. In the typical case of a set of parallel, horizontal lines, you supply a spacing value that has only a y-axis translation component. All lines have the same uncertainty in translation and rotation.

- Uniform uncertainties for all lines, in addition to individual poses for each line

  You arrange each line by supplying an arbitrary rigid pose (position and angle) for the line with respect to an implicit line arrangement coordinate system. You can also supply the uncertainty for the positions and angles of all lines.

- Individual pose and uncertainty factors for each line

The lines are arranged according to the rigid poses (position and angle) with respect to the implicit line arrangement coordinate system. The uncertainty in translation and rotation is also specified for each line.

The table below summarizes the various methods for specifying the spacing, pose, and uncertainty of lines in a line arrangement.

Method	Spacing	Pose	Uncertainty
1	uniform		uniform
2		arbitrary	uniform
3		arbitrary	arbitrary

Methods of specifying spacing, pose, and uncertainty for lines

Note: You should not attempt to specify a scale uncertainty value larger than 0.05 (which specifies a range of expected scale from 95% to 105% of the trained size).

Retraining the Tool

If you have changed the current key for a position or you want to verify a new line arrangement, you must retrain the OCV tool. You can retrain the tool by calling its **retrain()** function. This function retrains only the character and line information that has changed since the last training operation. It assumes that none of the alphabets contained in the arrangement originally used to train this tool have changed. If you have changed the alphabet, call the **train()** function.

Configuring Run-Time Parameters

Before you can run the OCV tool, you must configure its run-time parameters. This involves setting run-time parameters for each character position within a line, each line in a line arrangement, and the overall line arrangement.

Setting Character Position Parameters

You use the **ccOCVPosRunParams** object to describe the run-time parameters for a character position within a line. You set the following:

- A character position index that specifies the position in the line at which this character resides.
- An acceptance threshold.
- A confidence threshold.

For each character in a line, the OCV tool returns a score that indicates the match between the run-time character in the image and the character that is expected to be in that position. You can set an acceptance threshold for each character position. The accept threshold denotes the matching score below which the position fails verification. You can also set a confidence threshold for each character position. The confidence threshold specifies the amount that the matching score must be greater than the highest-scoring confusion character to pass verification.

Setting Line Parameters

After you define the run-time parameters for each character position in a line, you associate these with a line run-time parameters object. The **ccOCVLineRunParams** object describes the run-time parameters for a particular line of a line arrangement. These parameters include:

- The index of the line in an overall line arrangement.
- A set of character position run-time parameter objects for the character positions to verify within the line, specified in the order that the character positions should be verified. (Note blank characters are verified only after all other characters have been verified.)

Setting Tool Parameters

Before you can run the OCV tool, you must also set various tool run-time parameters. The **ccOCVRunParams** object encapsulates the information necessary to locate a line arrangement in a run-time image and to run the tool. There are three ways to create and configure a **ccOCVRunParams** object; these methods are described in the following sections.

In all cases, at run-time the OCV tool locates the line arrangement to verify based on the specified expected pose and uncertainty factors. The expected pose indicates the nominal pose of the fixture with respect to the image's client coordinates. The fixture offset describes the offset of the line arrangement to the fixture.

Supply a Vector of ccOCVLineRunParams

You can construct a **ccOCVRunParams** object by supplying a vector of **ccOCVLineRunParams** (one for each line) along with

- A timeout value, which indicates the maximum permissible length of time for the verification
- The expected pose of the fixture
- A fixture offset
- Pose uncertainty values

Supply a ccOCLineArrangement and Expected Pose

Instead of constructing and supplying a **ccOCVLineRunParams** object for each line in the arrangement, you can supply a **ccOCLineArrangement** together with the expected pose of the fixture. You also supply

- A timeout value, which indicates the maximum permissible length of time for the verification
- A fixture offset.
- Pose uncertainty values
- Run-time accept and confidence thresholds

If you use this method, the tool automatically initializes the run-time parameters for all character positions and lines to the values you supply.

In addition to supplying a **ccOCLineArrangement** together with the expected pose of the fixture, you can supply a **ccOCLineArrangement** together with the expected pose of a specific character. In this case, the tool automatically sets the expected positions of the other character and line positions based on the specific character position you define.

Setting Thresholds

The accept and confidence thresholds allow you to tune the sensitivity of the OCV tool to character and image variation. As you increase the accept threshold, the OCV tool will require a better match between the character in the run-time image and the trained character in the font. As you increase the confidence threshold, the OCV tool becomes more sensitive to confusing characters in the run-time image.

Note: The run-time confidence threshold is related to the font's training-time confusion threshold. As the font's confusion threshold is increased, fewer characters will be evaluated for possible confusion at run time.

If you set the accept threshold too high, the OCV tool will reject most or all of the character instances in the run-time image. If, as you are developing your application, the OCV tool fails to verify any characters, try reducing the accept threshold. Usually by reducing the threshold you will be able to determine why the characters in the run-time image are being rejected.

Similarly, if you specify too small a value for the confidence threshold, the tool will reject potentially confusing characters before evaluating their confusion with the found character. If, as you are developing your application, the OCV tool fails to indicate confusion with any characters with which you expect confusion to arise, try increasing the confusion

threshold. Usually by increasing the threshold, you will be able to determine why the characters in the run-time image are not being confused as they should be.

Specifying Uncertainty

By default, the OCV tool expects to find the characters to verify at exactly the positions you specify in the pattern. In most cases, run-time images will contain characters that vary slightly in size, position, and rotation.

Increasing the amount of expected uncertainty will allow your application to handle wider variations in run-time image appearance, but as the size of the search space increases, the amount of time required for verification increases as well. In almost all cases, however, setting some position, scale, and uncertainty range will improve the robustness of OCV.

Tool Uncertainty Limits

The OCV tool is designed to operate under a limited range of uncertainty values. The table below lists the maximum limits for rotational, scale, and translation uncertainty for line arrangements, lines, and characters.

Element	Uncertainty	Maximum Value
Line Arrangement	Angle	+/- 5°
	Scale	+/- 5%
	Translation	+/- 50% of the size of the first character in the first line
Line	Angle	+/- 5°
	Translation	+/- 50% of the line height
Character	Angle	+/- 5°
	Scale	+/- 5%
	Translation	+/- 50% of the character spacing

Uncertainty limits

Running the OCV Tool

After you have configured the OCV tool, you can use it to verify text. Running the OCV tool involves loading or acquiring an image that contains the text area you want to verify, passing the run-time parameters for character and line positions to the OCV tool, passing a tool results object to capture verification results, and invoking the tool's **run()** function.

Note: The scale component of the client coordinates of the image used to create the character models should be nearly equivalent to the scale component of the images supplied at run time. If the scales differ widely and you attempt to correct for this scale difference by specifying a large scale component in the expected pose, the tool may not work correctly.

Verification Results

The OCV tool stores the results of its verification run in a **ccOCVResult** object. The results object yields the following information:

- The measured pose of the line arrangement in client coordinates
- The measured pose of the fixture in client coordinates
- An overall verification score for the line arrangement
- The time required to run the tool
- The number of lines verified

- The verification status of the tool (true or false)
- A set of line result objects

To identify the characters in a line that failed, obtain the verification data for each character from the line results object. The OCV tool attempts to verify all the characters that you have specified in the run-time parameters. However, following a verification failure, the tool may not be able to locate some of the remaining characters. In this case, the tool reports as part of its results only characters verified up to and including the character that failed. If the timeout period for the tool is reached before the tool finishes running, the OCV tool returns no valid results.

Line Results

You can also examine the results reported for each line. The **ccOCVLineResult** object provides the following information about the verification of a line:

- The line index
- The measured pose of the line in client and line arrangement coordinates
- An overall verification score for the line
- The number of positions verified
- The verification status for the line (true or false)
- Character position results

In order for a line to pass verification, all characters within the line must be verified without confusion.

Character Position Results

If you require more detailed information about the verification results associated with a specific position within a line, you can examine the **ccOCVPosResult** objects contained in a line result object. The position results object provides you with information such as:

- The index of the character position in the line
- The key of the verified character
- The verification status for a character (verified, confused, or failed)
- The actual pose of the character in client and line coordinates
- A verification score for a character position, which indicates how well the character in the image matched the model of the expected character

Note: When the character status is *failed*, the score will always be zero unless there is a blank character in the current key set. In this case the score is nonzero.

Interpreting the Verification Status of a Character

For each character position, the OCV tool reports a verification status. This status can be one of the following:

- Failed
- Confused
- Verified

To evaluate the verification status for a character position, the OCV tool computes the match scores between the run-time image at that position and the models for each of the characters in the current key set. It uses the following criteria to determine the verification status:

- If the matching scores for all of the character models in the current key set are below the acceptance threshold, the tool reports a status of *failed*.
- If the highest match score is above the acceptance threshold, the character is either *verified* or possibly *confused* with another character. The OCV tool compares this score to the scores of characters with which it can potentially be confused.
    - If the difference between the highest match score and the score for each confusion character is greater than the confidence threshold in every case, the tool reports a status of *verified*.
    - If the difference between the highest match score and the score for any confusion character is less than the confidence threshold, the tool reports a status of *confused*.

Note: You can obtain the keys and scores of all characters that were confused with the found character.

False Matching of Blank Characters

If a character position in a line arrangement is defined with a blank character among its current keys, and if that character position is not blank in the run-time image and does not match any of the other current keys, this character position should fail. However, depending on the character in the image, it is possible for the OCV tool to incorrectly mark it as *verified*. This occurs when the illegal character in the image occupies very few pixels, thus containing mostly blank space which produces a high correlation and a high matching score with the blank key. The table below shows some typical blank key scores for the smallest characters.

Run-time character	Score	Run-time character	Score
(-) minus sign	0.423	(,) comma	0.276
(.) period	0.422	(+)	0.106
(‘) apostrophe	0.382	(\)	0.078
(:) colon	0.312	0	0.021

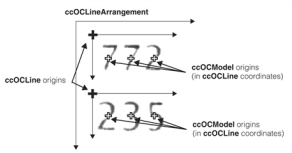
Blank character key scores

If you encounter this situation in your application and you have set your acceptance threshold too low, a small character such as a period in an illegal position can be verified as a blank. Note that for blank character matches, the OCV tool does not check for confusion as it does for all other characters. If the blank score is above the acceptance threshold it marks the position as *verified* and moves on. For this reason, you will never see a confusion status after a matched blank. Note however, that if the tool finds a match on a small character such as a period, its possible for the period to be confused with a blank if the confidence threshold is set too low.

Understanding How Positions are Reported

The OCV tool reports position information for the overall line arrangement, the individual lines within the arrangement, and for the characters within each line. The position values that the tool returns are determined both by where the characters are located in the image and by the expected pose and uncertainty information that you supply.

The figure below shows a simple arrangement comprising two lines, each of which contains three characters.



An arrangement of characters and lines

The origin of each character is defined in the coordinate system of the line that contains it, while the origin of each line is defined in the coordinate system of the overall line arrangement.

When you attempt to verify the line arrangement shown in the figure above, you must supply the expected position of a fixture for the line arrangement in the client coordinates of the run-time image. (As described in the section *Setting Tool Parameters* on page 503, you can specify this in several different ways.) You also specify the maximum uncertainty for

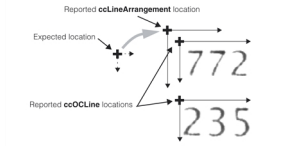
- The rotation and translation of the fixture for the overall line arrangement
- The rotation and translation of the lines within the line arrangement
- The rotation and translation of the character models within the line

The figure below shows an example of a run-time image with the expected location of the line arrangement shown in the figure above.

Expected location in a run-time image

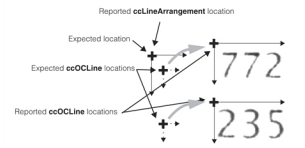
Depending on the positional uncertainty ranges you specify, the tool could report the locations of the found line arrangement, lines, and characters in several different ways.

For example, if you supplied a positional uncertainty range of plus or minus 10 for both the fixture for the line arrangement and the line positional uncertainty, the tool could report the results shown in the figure above as the translation of the fixture for the line arrangement, with the lines themselves located exactly as expected. This is shown in the figure below.



Reported translation in line arrangement, no translation in lines

Alternatively, the tool could report that the fixture for the line arrangement was found at exactly the expected location, but the individual lines were translated from their expected locations with respect to the origin of the fixture for the line arrangement. This is shown in the figure below.



Report translation in lines, no translation in line arrangement.

The examples shown in the three figures above is limited to uncertainty in the fixture for the line arrangement translation and line translation. The same alternatives are also present for rotation, and for the expected locations of individual characters within a line.

Normally, when the tool is confronted with the situation described above, the tool returns the “simplest” result. That is, translation is first assumed to be at the level of the overall line arrangement, than at the level of individual lines, and finally at the level of individual characters.

Note: The positions that the OCV tool reports for line arrangements, lines, and characters are always constrained by the expected positions and uncertainty ranges that you specify. The tool never reports a position that lies outside of the uncertainty range that you specify, except when you request the position of a character in the client coordinates of the input image (**ccOCVPosResult::clientPose()**), in which case the actual position of the character is returned.

OCVMax Tool

The OCVMax tool performs *optical character verification (OCV)*, a process for verifying that one or more character strings in an image contain the expected characters at each position. Vision applications typically use OCV to verify lot codes, date codes, expiration dates, and other information printed on products or packaging.

Some Useful Definitions on page 508 defines some terms you will encounter in this chapter.

OCVMax Tool Overview on page 508 outlines how the OCVMax tool works.

Using the OCVMax Tool on page 509 describes the techniques that you use to implement an OCVMax application.

Differences from the OCV Tool on page 523 summarizes the differences between the OCVMax tool and the OCV tool.

Some Useful Definitions

This section defines some terms and concepts used in this chapter.

accept threshold: If the **matching score** is below this level the character at the specified position fails verification.

arrangement: A sequence of character lines in space.

confidence threshold: If the difference between the **matching score** for the position and the highest scoring confusion character is greater than the confidence threshold, the position passes verification; otherwise, the verification status of the position is confused.

confusion score: A number in the range 0 through 1 that indicates the probability that two characters can be confused with one another during optical verification. A value of 0 means that the characters will never be confused with each other; and a value of 1 means that they will always be confused.

confusion threshold: If the computed confusion score for a character pairing is greater than the confusion threshold, the characters are easily confused with each other during optical verification.

degrees of freedom: A transformation type that changes the appearance of a character and can be characterized by a single numeric value.

matching score: The score generated when the OCVMax tool measures the match between the run-time character image for a position and the model image for a character expected to be in that position.

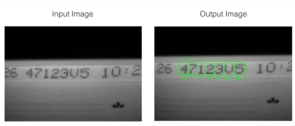
optical character verification (OCV): The process of verifying that the characters in a text area are the expected characters.

paragraph: A string or a set of strings containing characters using the same font.

OCVMax Tool Overview

Use an OCVMax tool to verify that characters printed on labels or displayed on a panel are readable.

The figure below shows an example image where an OCVMax tool has been used to verify a string composed of a 5x7 dot-matrix font.



Using the OCVMax tool to verify a string composed of a 5x7 dot-matrix font

A configured OCVMax tool can tolerate the following types of image variations:

- Lighting
- Image quality
- Contrast
- Font stroke width
- Position, rotation, and scale for the overall string(s) of characters
- Character-to-character position, rotation, and scale

In general, using an OCVMax tool requires your vision application to complete the following tasks:

1. Acquire a training image containing the string(s) of characters you want to verify.

   See the *Acquiring Images* chapters of the *CVL User's Guide* for more information on acquiring images.

2. Specify a font file for the characters that appear in those strings.

   See the section *Font Files* on page 509 for more information on font files.

3. Specify the string(s) based on the goal of the application.

   See the section *Paragraphs* on page 511 for more information.

4. Render an example of the string(s) using the font described by the font file.

   See the section *Font Rendering* on page 512 for more details on rendering a string.

5. Allow the tool to use PatMax technology to determine the best possible search parameters for reliably locating the string in successive run-time images.

   See the section *Degrees of Freedom* on page 514 for information on the ways a character string can change in appearance between the input image and a run-time image. See the section *Training and Tuning* on page 516 for more information on OCVMax tuning.

6. Try the configured OCVMax tool on successive test images to ensure it reliably verifies the characters in each string. As you test the tool you can modify several search parameters as necessary to compensate for image-to-image variations in the appearance of the strings.

7. Verify the OCVMax tool works to locate and verify the strings in your test images, and then reduce the range of various search parameters in an effort to improve the performance time for each image. Use the hints provided by the OCVMax tool to enable, disable, or reduce various search parameters until the tool performs successfully within the best possible time frame.

Using the OCVMax Tool

This section describes how to use the OCVMax tool.

Font Files

To configure an OCVMax tool you need a font file that already defines the layout of each character in the font, also known as *character keys*. You can specify any font file for use with an OCVMax tool, which supports all Western-language TrueType ASCII and 16-bit Unicode character fonts (21-bit Unicode fonts are not supported). Other font types might work with the OCVMax tool, but you must experiment with acquired images and OCVMax tool parameters to determine how well they work.

Font information to be used by the OCVMax tool is contained in *paragraphs* that are part of an *arrangement* you supply to the tool.

You use the **ccOCVMaxParagraph::font()** function to specify the font for a paragraph or obtain the font information for a paragraph.

Character strings applied by laser jet (also known as video jet) or thermal printer likely use a common font. You must acquire the font file separately. In order to locate the correct font file for your vision application, Cognex makes the following suggestions:

- Contact your print equipment vendor.

  The vendor should be able to provide you with the name of the font the printing equipment uses, and perhaps even the font file itself.

- If you know the name of the font, search the \WINDOWS\Fonts directory on your PC.

  If your print equipment uses a TrueType font, you can preview all TrueType fonts by using the font dialog box of Microsoft Word.

  Note: Microsoft Windows prevents users from directly opening font files in the \WINDOWS\Fonts directory. If you need to use a font from \WINDOWS\Fonts, first copy it to any other local directory.

- Generate your own font file by using the Image Font Extractor, available by choosing Start->All Programs->Cognex->CVL->Utilities->Image Font Extractor.

  To generate your own font file you need an image containing examples of the characters you want to verify.

- Try the http://www.myfonts.com/WhatTheFont web site, which is an online source for finding, trying, and buying fonts. The web site boasts 41,680 fonts with search tools that allow you to find and buy the right font to match your printed strings.

An OCVMax tool can use the following font file types:

Font Extension	Font Type
.ttc	TrueType collection font file (a font file that contains multiple TrueType fonts)
.ttf	TrueType font
.cst	Domino stroke font
.gcg	Imaje raster font
.ffn	The OCVMax tool supports the following Markem fonts: • 16 high x 10 wide (hi_res16.ffn) • 7 high x 5 wide (hi_res7.ffn) • 5 high x 5 wide (char_5_5.ffn)
.xcl	VideoJet raster font
.fnt	Zebra raster font (many other font files may also use this extension)
.bdf	Glyph Bitmap Distribution Format font
.ocf	Cognex image font file
.ocm	Cognex extended image font file

Diacritical Characters in Unicode

The OCVMax tool does not support the use of any 'combining' characters, such as Unicode combining diacritical characters. The OCVMax tool assumes that each character position is associated with a single character code or Unicode 'code point'. If you need to verify characters with diacritical marks, use a precomposed Unicode character code.

Using Image-Based Font Files

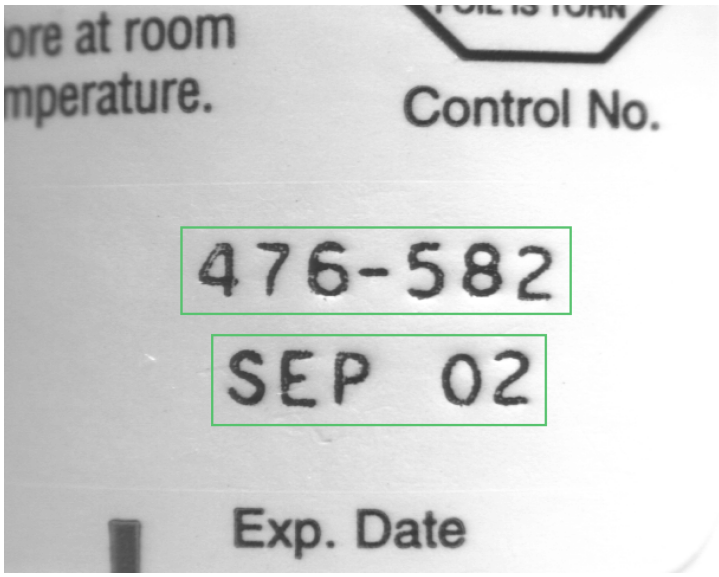
If you use an .ocf file generated by the Image Font Extractor, the 32-bit character key value for each character in the font file is defined as follows:

- The high-order 16 bits give the Unicode code point of the first character of the character position's name.
- The low-order 8 bits give the character instance number, if the font has multiple instances of the same character.

- Bits 8-15 are an arbitrary value set so that the 32-bit key value is unique within the font. Usually these bits are set to 0x00.

Paragraphs

As you configure an OCVMax tool, you must specify a *paragraph* or set of character strings you want the tool to verify. A paragraph can contain a single string or a set of strings. For example, the figure below shows an image of a single paragraph containing two strings.



An image of a single paragraph containing two strings

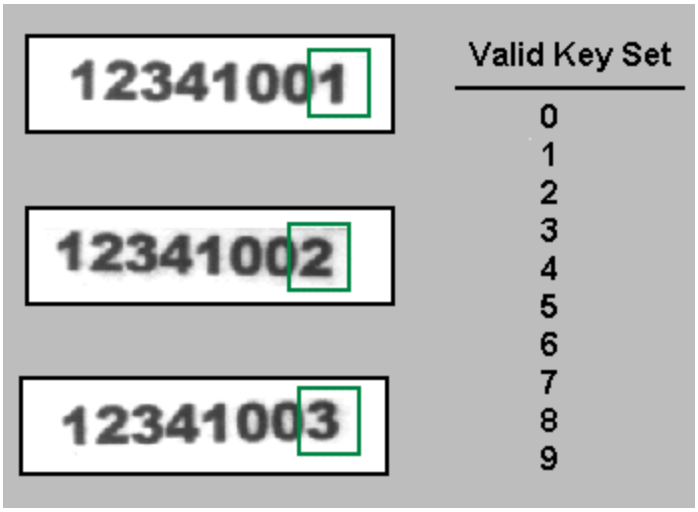
In addition, a single OCVMax tool can contain multiple paragraphs. A single paragraph can only reference a single font, but multiple paragraphs in the same OCVMax tool can reference different fonts. You specify the number of paragraphs an OCVMax tool will contain and the number of character strings in each paragraph.

You use the **ccOCVMaxLine** object to construct a line, the **ccOCVMaxParagraph** object to construct a paragraph, and the **ccOCVMaxArrangement** object to construct an arrangement you want the OCVMax tool to verify.

Key Sets and Wildcards

For every individual position within a character string, an OCVMax tool uses a *key set* to specify the individual characters it expects to verify. A character string of 10 characters, for example, uses 10 key sets.

For character strings that remain constant, each key set contains a single character key. When you want to verify a character string where one or more character keys at a specific position can vary between run-time images, you must create a key set with *wildcards*, or multiple character keys. For example, the figure below shows three character strings that can all be verified with the same OCVMax tool, since the key set at the last position has been configured with all digit characters.



Three character strings that can all be verified with the same OCVMax tool

Be aware that specifying a mix of wildcard characters that contain both blank and non-blank characters is not recommended and can seriously impact the performance of an OCVMax tool.

Train-Time Key Sets and Run-Time Key Sets

An OCVMax tool supports a *train-time key set* as well as a *run-time key set*. Use a train-time key set to specify all the character keys possible at a certain position, and a run-time key set to specify the expected character key at that position in the next run-time image. For example, a vision application might verify a string of numbers where the values at one or more positions within the string increase incrementally. As you configure an OCVMax tool, you would specify a train-time key set with all the numerical character keys, 0 - 9, at those positions. As the application operates, you would use a run-time key set of the next character key you expect based on the last image analyzed by the OCVMax tool. Using the run-time key set generally improves the performance speed of the tool versus having the tool compare the current character with every character in the train-time key set.

You use the **ccOCVMaxArrangementSearchKeySets**, **ccOCVMaxLineSearchKeySets**, and **ccOCVMaxParagraphSearchKeySets** objects to specify what key sets should be searched for at runtime.

Optical Character Recognition

The OCVMax tool is not intended to perform OCR (Optical Character Recognition). Specifying wildcards for every character in a string may result in poor performance.

Font Rendering

The OCVMax tool uses PatMax technology to search for the character strings in each acquired image. After you specify the necessary key sets, you can graphically generate (render) an example of each string you need to verify into an outline, which is a shape that can be used to produce a CVL graphics object. You can overlay the generated string graphics on a tuning image to visualize whether the font parameters and scale look correct for tuning.

Note: CVL does not use an image for training, only for tuning.

You use the **ccOCVMaxArrangement::render()** function to render an arrangement into an image with a given position or to render an arrangement into a shape that can be used to create overlay graphics.

The position you specify provides the PatMax algorithm a logical starting position for each search. The performance and the ability of the PatMax algorithm to tune the OCVMax tool with good search parameters for reliably locating each character in successive run-time images depends on the following factors:

- How accurate the provided position of the string graphics is; that is, how accurately the the string graphics have been positioned over the string you will need to verify in the tuning image.
- How good the initial values are for the size, rotation, and character spacing of the string graphics.

Graphically rendering an example of each string you want to verify is not mandatory, but generally improves the performance speed and reliability of the OCVMax tool.

For example, the figure below shows an image containing a single string.

Image containing a single string

The figure below shows the same image after a user has rendered a string over the corresponding characters in the tuning image.

Rendered string positioned over the corresponding characters

As you render a character string over the tuning image, you might notice the character keys in the image show slightly different characteristics than the character keys in the font you specified, even though you chose the correct font. For example, the printed strings in your images might use a dot-matrix font where the dots are not symmetrical in shape, or are slightly larger in scale than specified by the font.

You can compensate for these types of character key differences with the following advanced font parameters:

- Additional Character Spacing

  You can specify additional spacing, in pixels, between character keys in both the horizontal and vertical direction.
- Additional Line Spacing

  The tool allows you to specify additional spacing, in pixels, between lines of character keys when you specify a multi-line string.
- Character Features

Primarily for dot-matrix fonts, you can specify the difference in scale between dots in the image and the dots specified by the font you use, or increase the stroke width in non dot-matrix fonts.

Be aware, however, that if you enable the tuning feature of an OCVMax tool, the tool itself can set good starting values for these advanced parameters, although you can always choose to change their values later to improve reliability or execution speed.

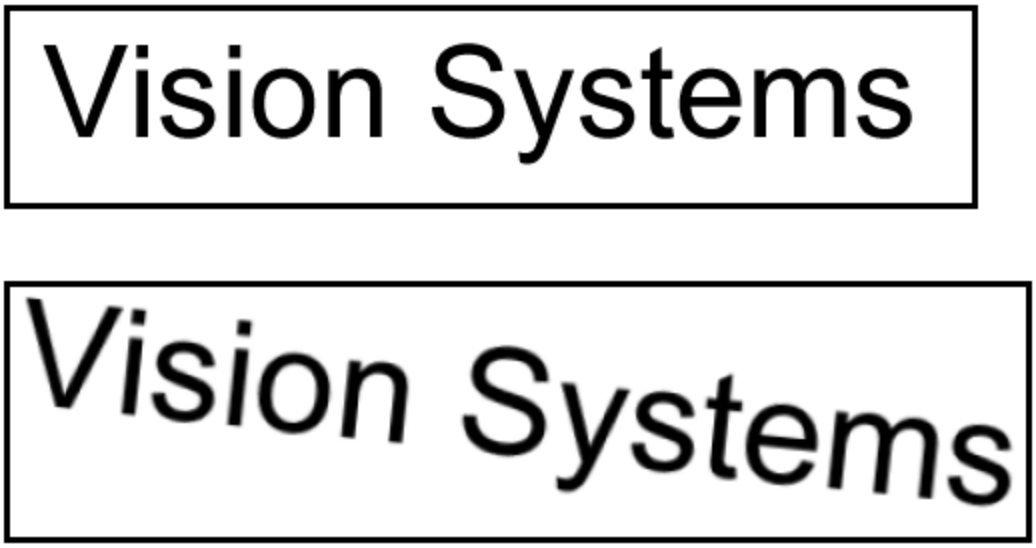
Degrees of Freedom

The appearance of characters can change in a variety of ways from one image to the next. Each of these types of transformation is called a *degree of freedom*, and can be characterized by a single numeric value, such as the amount of rotation a character undergoes from one image to the next. See examples for transformations in the table below.

Degree of Freedom	Example
X Shift	
Y Shift	
Angle	
Uniform Scale	
Scale X	
Scale Y	
Shear	

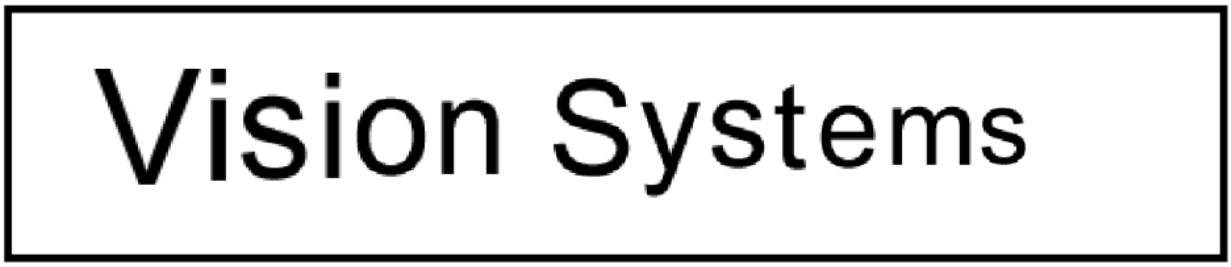
Degrees of freedom

Usually, changes in degrees of freedom affect the entire character string, not individual character keys. For example, the figure below shows an entire character string and a transformed version of it that has undergone a change both in Angle (Δ = 6 degrees) and Uniform Scale (Δ =1.15).



An entire character string that has undergone both a change in Angle and Uniform Scale

Depending on your print equipment or surface material, however, your character string can manifest a particular degree of freedom on a character-to-character basis. For example, the following figure shows a character string where each subsequent character key in the string suffers a decrease in uniform scale:



Subsequent decrease in uniform scale

For character strings such as this, the OCVMax tool must know about the change in uniform scale on a character-to-character basis.

As you tune an OCVMax tool, you must acquire a training image of the character string(s) you want to verify, and then create examples of each string using character keys from a matching font file. Using this training image and rendered character strings, an OCVMax tool can use its tuning feature to automatically detect which degrees of freedom you should enable for initial testing against the images you acquire later.

The tuning feature uses PatMax technology to test a wide range of values for each degree of freedom to determine the best match between the character string you want to verify and the underlying example image. See the section *Training and Tuning* on page 516 for more information on OCVMax tuning.

Once the tuning process completes, the OCVMax tool will have a set of parameters that should provide a good basis for the final set of parameters you will use when you deploy your application. As you test the OCVMax tool on different sample images, the OCVMax tool can make additional recommendations for modifying the range for various degrees of freedom, as well as hints for other search parameters. Be aware, however, that the OCVMax tool will not make recommendations for any degrees of freedom you disable.

You can enable or disable degrees of freedom using the following functions:
ccOCVMaxSearchRunParams::zoneEnable()
ccOCVMaxTrainParams::zoneEnable()

Note: Only Correlation Registration Mode (see *Correlation Registration Mode* on page 522) uses the zones specified in the training parameters. Standard Registration Mode ignores any zone settings in the training parameters.

Although you can disable the tuning feature of an OCVMax tool and set which degrees of freedom you want the tool to consider, this can cause the tool to require a great deal of time to determine which settings provide the best results. Cognex recommends you use the tuning feature of an OCVMax tool to generate the best set of parameters as you move toward testing the tool on various sample images.

In general, the number of degrees of freedom you enable increases the amount of time the OCVMax tool requires to sufficiently analyze an image and perform verification. Disabling additional degrees of freedom should improve performance time, at the risk of failing to verify particular character keys because of the degrees of freedom they might exhibit.

Be aware that simply enabling a degree of freedom does not improve the ability of an OCVMax tool to reliably verify a character key in successive run-time images. As the OCVMax tool attempts to verify a particular character key in a string, the degrees of freedom currently enabled determine the overall strength of the match. For example, the same character key can be perceived to be a better match with the Uniform Scale degree of freedom disabled rather than enabled.

This is especially true when a particular degree of freedom is enabled but the found character key exhibits a transformation outside the current range. For any enabled degree of freedom, the set range must encompass the variety of transformations a character key can undergo in order to prevent that degree of freedom from having a negative impact on the ability of the OCVMax tool to verify that character key.

Training and Tuning

An OCVMax tool must be trained in order to reliably locate the characters in the strings you want to verify. CVL uses PatMax technology to train the OCVMax tool based upon an actual image of the string(s).

You use the tool's **train()** function to train the tool.

Before you can tune an OCVMax tool, you must acquire an image of a character string showing the characteristics you expect the tool to encounter when you deploy your vision application to the production environment.

While training, you specify the *clientFromImage* parameter of the tool's **train()** function to supply the coordinate transformation that will render the font into characters that are approximately the same size as those expected to be found in runtime images.

As you train an OCVMax tool, you select the amount of clutter (noise or false image data) that the training image contains. Image clutter can come from a variety of sources, including irregular lighting and camera optics. Well-lit images with good contrast typically have low clutter level values. In practice, higher quality images produce better and faster results. You use the **ccOCVMaxTrainParams::scoreMode()** function to select the image clutter level the tool should consider when computing scores for characters.

Note: **ccOCVMaxTrainParams::scoreMode()** applies only to standard registration mode.

In addition to training, you can allow the OCVMax tool to tune certain parameters automatically (using the tool's **tune()** function) by examining the features of the character string in the acquired image.

Tuning an OCVMax tool allows it to use PatMax technology to experiment with a variety of parameters and determine the best settings based on the degrees of freedom the characters in your strings are likely to experience.

The amount of time an OCVMax tool requires for tuning will vary depending on the number of characters and the type of font you use. Some character strings can require a significant amount of time to tune an OCVMax tool for successful use. For this reason, you can use the **ccOCVMaxProgress** object to check how far along any single tuning or training operation has proceeded.

If you provide a logical starting position for the search process by specifying the *startPose* parameter while using the tool's **tune()** function, the search process begins with the assumption that the string will be found very close to the given position. If this is indeed the case, the tool can tune much faster than when it needs to search the entire image first.

Tuning is optional. Although you can disable automatic tuning as you train an OCVMax tool, it can be difficult for you to determine the best settings for various image and character-to-character parameters that allow the underlying PatMax technology to locate the strings in successive images.

The tuning operation is not always successful. In some cases, the OCVMax tool can report a failed attempt at tuning. In general, tuning fails for one of two reasons:

- The string you rendered to train the OCVMax tool does not match the actual string that appears on the training image.
- The tool graphics for the string are not close enough for PatMax technology to locate the actual string in the training image.

Finally, an OCVMax tool should be able to reliably locate and verify the characters in valid character strings once the tuning operation has completed. Be aware, however, that the time required to perform a verification can be excessive. By manually adjusting various search parameters after the tuning operation has assigned good starting values, you can typically reduce the amount of time required by the OCVMax tool to locate and verify the character strings in the images you acquire.

Scores and Thresholds

An OCVMax tool generates a variety of scores and compares them against assorted thresholds in an effort to verify each character in your character strings. See the following two sections for more information.

Matching Scores and Accept Thresholds

For every character key it examines, an OCVMax tool generates a matching score to indicate how well the actual character key in the image matches the expected character key for this position, taking any key sets into consideration. Matching scores range from 0.0 to 1.0, with 1.0 the score for a perfect match.

The tool compares this matching score against an accept threshold, and gives the character key a preliminary passing result if the matching score exceeds the accept threshold. By default, the OCVMax tool uses an accept threshold of 0.5, but you can raise or lower the value as necessary for your vision application.

The accept threshold allows you to vary the sensitivity of the OCVMax tool to character and image variation. As you increase the accept threshold, the OCVMax tool will require a better match between the expected character key and the actual character key in the run-time image.

Raise the accept threshold when your vision application acquires good images of the character strings you want to verify, with character keys of consistently good print-quality, and that exhibit little change in various degrees of freedom. A high accept threshold can improve the execution speed of the OCVMax tool. If you set the threshold too high, however, the OCVMax tool will reject most or all valid character keys in the image.

Lower the accept threshold when your production environment cannot ensure a constant appearance for each character key or when character keys can undergo a change in one or more degrees of freedom. If, as you are developing your application, the OCVMax tool fails to verify correct character keys, try lowering the accept threshold.

An OCVMax tool uses the following two accept thresholds, both applying to character scores:

- The accept thresholds specified in the character key search parameters.
- The accept thresholds specified in either the image search parameters or the start pose search parameters (depending on which overload of the tool's **run()** function is called).

Both accept thresholds should typically be set to the same value.

If you enable the tuning feature, an OCVMax tool can generate a good value for each accept threshold automatically. You can further modify the accept thresholds to improve performance or improve reliability.

You use the **ccOCVMaxSearchRunParams::acceptThreshold()** function to adjust accept threshold levels.

Confusion Scores and Confusion Thresholds

When you train an OCVMax tool, the tool creates a set of confusion scores for every character key you enable in the font you use. Confusion scores represent the amount of similarity between each pair of character keys, and are generated by comparing character keys at the same position, angle, and scale. An OCVMax tool stores the confusion scores in a confusion matrix, as shown in the following example.

	Char	0	1	2	3	4	5	6	7	8	9
▶	0	0.00	0.20	0.09	0.48	0.00	0.27	0.64	0.00	0.54	0.68
	1	0.00	0.00	0.00	0.00	0.43	0.00	0.00	0.00	0.00	0.00
	2	0.22	0.00	0.00	0.30	0.00	0.00	0.05	0.05	0.22	0.25
	3	0.63	0.00	0.31	0.00	0.00	0.45	0.73	0.00	0.83	0.69
	4	0.00	0.44	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00

Confusion matrix example, with confusions scores above 0.5 highlighted in red

The character keys across the top of the confusion matrix represent all the enabled character keys in the current font, while the character keys down the left-hand column represent all the character keys in the string you want to verify, in numerical order. For example, this confusion matrix could represent the string "322104" in a font where only numbers are enabled. If you configure an OCVMax tool to verify multiple paragraphs, each paragraph generates its own set of confusion scores.

To read the confusion score between any two character keys, start in the left-hand column. For the previous confusion matrix, comparing the character key "3" to the character key "5" reveals a confusion score of 0.45, while comparing the "3" to the "8" reveals a confusion score of 0.83. The table may not always contain symmetric values when comparing the value at (i, j) with the value at (j, i). For example, the confusion score comparing "0" to "3" is not the same as comparing "3" to "0".

Scores that appear in red in the confusion matrix example in the figure above represent confusion scores that exceed the confusion threshold value of 0.5, and indicate pairs of character keys where the potential for confusion can be high. As the OCVMax tool attempts to verify each character key, the tool performs some extra processing to rule out potentially confusing character keys. Using the previous confusion matrix as an example, an OCVMax tool attempting to verify a character key in the string against the character key "0" will generate a matching score for the character key "0" as well as the character keys "6", "8", and "9". The matching score for "0" must exceed the matching score for the other potentially confusing character keys, and it must exceed these scores by a margin set by a new threshold, the confidence threshold. Typically, the confidence threshold is set to a low value, such as 0.1.

If the matching score a character key receives does not exceed all other potentially confusing scores by the margin set by the confidence threshold, the OCVMax tool does not verify the character key and instead reports the character key is Confused.

As you decrease the confusion threshold, the OCVMax tool generates matching scores for more potentially confusing character keys and compares them to the matching score of each character key it attempts to verify. This yields better verification results, but it requires more time for the tool to execute and it can increase the chance that character keys will be erroneously rejected due to confusion. As you increase the confusion threshold, the tool generates fewer matching scores for potentially confusing character keys, resulting in faster execution times but increasing the chance that the tool will verify a character key incorrectly.

The confidence threshold can also be adjusted from its default low value. Increase the confidence threshold if you become concerned that the OCVMax tool verifies character keys incorrectly; for example, verifying that a character key is a "3" when it appears as an "8" in your acquired image.

You use the **ccOCVMaxParagraphRunParams::confusionThreshold()** and **ccOCVMaxParagraphRunParams::confidenceThreshold()** functions to adjust confusion and confidence threshold levels.

Early Accept Thresholds

You can also specify early acceptance or early fail thresholds.

If the percentage of keys in the line that match (that is, are verified or confused) is above the early acceptance threshold, no further searching for the line is performed. This speeds up the search process when the user knows that there will not be spurious characters in the run-time image (for example, when it is known that the only text in the image will be the string to be verified). You use the **ccOCVMaxRunParams::earlyAcceptThreshold()** function to specify the early accept threshold.

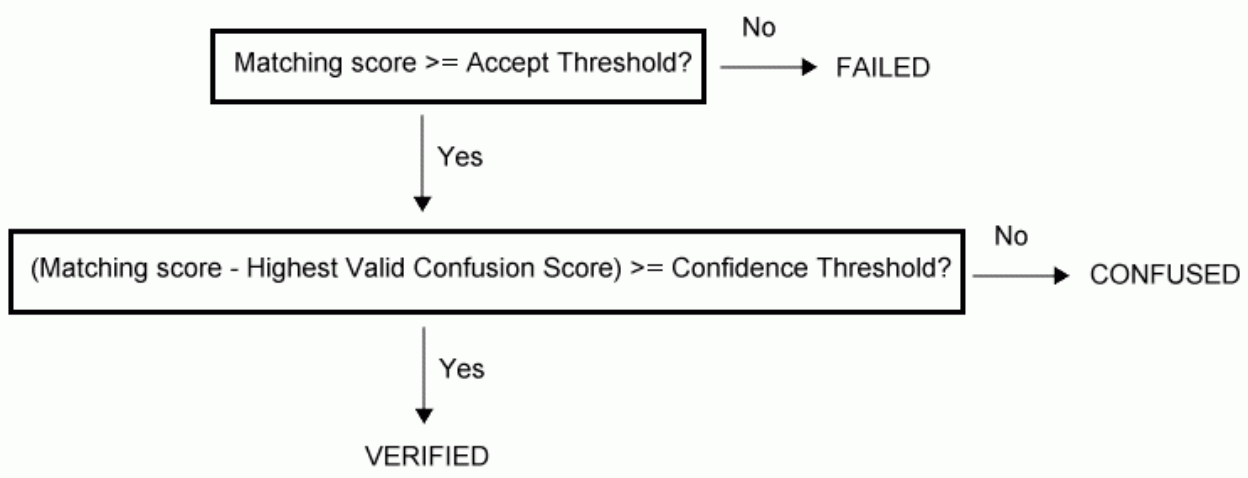
If the percentage of keys in the line that match (that is, are verified or confused) is below the early fail threshold, the section of the image containing the potential line will not be considered any further. This speeds up the search process when the user knows that the image should be clean, with no missing or badly damaged characters allowed. You use the **ccOCVMaxRunParams::earlyFailThreshold()** function to specify the early accept threshold.

Character Verification

To verify a single character key in any character string, the OCVMax tool performs the following steps:

1. The tool generates a matching score comparing the expected character key to the character key being verified. See section *Matching Scores and Accept Thresholds* on page 517 for more information. The matching score must exceed an accept threshold in order to continue verification. If the matching score falls below the accept threshold, the character key fails verification immediately.
2. Using the confusion matrix, the OCVMax tool compares the matching score against the matching scores generated for all potentially confusing character keys. See section *Confusion Scores and Confusion Thresholds* on page 518 for more information. If there are no confusion scores higher than the confusion threshold, the character key passes verification immediately.
3. The tool subtracts the highest potentially confusing score from the matching score, and compares that resulting value against the confidence threshold.If the value is greater than or equal to the confidence threshold, the character key passes verification; otherwise, the tool reports the character key being identified is currently confused with another character key.

The verification process can be described graphically with the following flowchart:



Verification process flowchart

You use the **ccOCVMaxLineResult**, **ccOCVMaxParagraphResult**, **ccOCVMaxPositionResult**, and **ccOCVMaxResult** objects to obtain verification results. You can also use the **ccOCVMaxPositionResultStats**, **ccOCVMaxResultDOFStats**, and **ccOCVMaxResultStats** objects to accumulate statistics over sets of results; such statistics can be helpful for setting DOF ranges and score thresholds or for process control.

Run Timeout

As you configure an OCVMax tool, you can set a run timeout value, which determines the amount of time in milliseconds in which the OCVMax tool is allowed to return the best result for the verified character keys in your string(s).

The OCVMax tool considers the run timeout value in any attempt to locate and verify each character key in the string. Most strings will fail verification if the run timeout value is too small. If you allow the OCVMax tool to tune itself with good values for various search parameters, the tool will generate a good starting value for the run timeout.

You use the **ccOCVMaxTuneResult::runTimeout()** function to get a run timeout value that is sufficiently large to allow a run to complete and set the run timeout value for the OCVMax tool.

In addition, although a string can pass verification with a particular run timeout value, enabling a larger run timeout value can generate more accurate search scores for each character key in the string. For some images, more accurate search scores can mean the difference between a verified character string and one that fails verification. In general, you should use as large a run timeout value as you can allow for each inspection of your vision application.

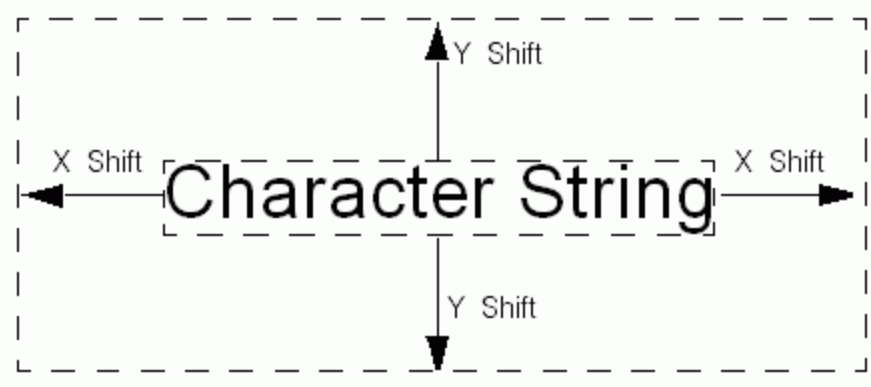
In some cases an OCVMax tool can return a successful result while exceeding the set run timeout value. If your vision application requires that the OCVMax tool complete its analysis within a specific time allotment, set the run timeout value to a few milliseconds less to accommodate for this type of situation.

You can use the **ccOCVMaxResult::timeoutOccurred()** function to check if timeout occurred.

Search Modes

For each image you acquire, the OCVMax tool must locate the character string (or strings) you want to verify. The OCVMax tool supports the following two possible search modes:

- Start Position + Shift Uncertainty:
  The OCVMax tool can be configured to start a search for each character string based upon where the string was found in the training image, and extending outwards along the x-axis and y-axis by a set of amount of pixels. The OCVMax tool uses the parameters *X Shift* and *Y Shift* to define the borders of the search area, as shown in the following figure.



X Shift and Y Shift to define the borders of the search area

Of the two search modes, this mode offers a faster search algorithm. Cognex recommends using Start Position + Shift Uncertainty whenever possible.

For this mode, you use the overload of the **ccOCVMaxTool::run()** function that specifies the *startPose* parameter.

You use the **ccOCVMaxSearchRunParams::xyUncertainty()** function to set the uncertainty of the starting pose in the x and y dimensions, in client coordinates.

You can also use the Start Position + Shift Uncertainty mode when using the tuning feature of the OCVMax tool by using the overload of the tool's **tune()** function that specifies the *startPose* parameter. Once tuning is complete and you are testing the current set of OCVMax tool parameters on sample images, you can choose to switch to a different search mode as appropriate for your application.

- Whole Image:
  You can configure the OCVMax tool to search each entire image for the strings you want to verify.

  For this mode, you use the overload of the **ccOCVMaxTool::run()** function that does not specify the *startPose* parameter.

  Searching the whole image requires more time. Cognex recommends you use this option only when the position of the character strings can vary across the entire image.

Search Parameters

The OCVMax tool uses a variety of search parameters to locate the character strings in the images you acquire. The tool uses one set of search parameters to specify the full range of DOF and positional variation that can occur for any character in the arrangement, and the tool uses another set of search parameters to specify the relative range of DOF and position variation that can occur between one character and the character adjacent to it. The *full search* parameters are called *image search parameters* when using an overload of the tool's **run()** function that searches an entire image window and are called *start pose search parameters* when using an overload of the tool's **run()** function that takes a start pose. The relative search parameters are called *character key search parameters*. For simplicity, this document may sometimes use the term *image search parameters* instead of *full search parameters*.

For example, for strings where all the characters are printed consistently, for example, they all appear in a straight line at the same scale, one would typically use image search parameters that specified the range for any character, but character key search parameters that have all the DOFs disabled (because given, for example, the scale of one character, the scale of the adjacent character is guaranteed to be the same).

In almost all cases, the character key search parameters will specify a smaller range than the full search parameters. Also, in almost all cases, the accept thresholds specified in the full search parameters and the character key search parameters should be the same.

By enabling the tuning feature, an OCVMax tool can generate good starting values for these search parameters. You can enable, disable, and alter these values as necessary to improve the ability of the tool to locate and verify good character strings, or to reduce the time necessary for the tool to function successfully. See the section *OCVMax Optimization* on page 523 for optimization suggestions.

You use the **ccOCVMaxSearchRunParams** object to specify search parameters.

Image Search Parameters and Start Pose Search Parameters

An OCVMax tool uses the following image search or start pose search parameters to locate the character strings you want to verify:

Parameter	Description
Accept Threshold	The tool allows you to set an accept threshold to gauge the similarity between the characters in the string and various features in the image that could resemble the characters. See the section *Matching Scores and Accept Thresholds* on page 517 for more information.
Degrees of Freedom	The tool allows you to set various values for the degrees of freedom the characters can undergo between successive images. These values should represent the maximum range of these values for any character in the image. See the section *Degrees of Freedom* on page 514 for more information on degrees of freedom.

Character Key Search Parameters

An OCVMax tool uses the following character key search parameters to specify the relative appearance of adjacent characters in the string:

Parameter	Description
Accept Threshold	The tool allows you to set an accept threshold to gauge the similarity between the characters in the string and various features in the image that could resemble the characters. See the section *Matching Scores and Accept Thresholds* on page 517 for more information.
Degrees of Freedom	The tool allows you to set various values for the degrees of freedom the characters can undergo between one character and an adjacent character. For example, if the scale of one character is 1.05 and the scale of an adjacent character is 1.1, then the relative range is [1.05/1.1, 1.1/1.05] or [0.955, 1.048]. Similarly, if the angle of one character is 10 degrees and the angle of an adjacent character is 17 degrees, then the relative range is [10-17, 17-10] or [-7, 7]. **Note**: In character key search parameters, the low and high values should be multiplicative inverses of each other for scale DOFs, and the low and high values should be additive inverses of each other for angle and shear DOFs. See the section *Degrees of Freedom* on page 514 for more information on degrees of freedom.

Correlation Registration Mode

An OCVMax tool allows the character strings in the images you acquire to exhibit a wide range in various degrees of freedom from one image to the next. In addition, using Standard Registration Mode allows your strings to be verified in an array of lighting conditions, as well as allowing the tool to be trained despite the level of image noise in the background.

An OCVMax tool also supports the Correlation Registration Mode, which can be used in almost all the same situations as Standard Registration Mode, except that it cannot handle nonlinear brightness changes in the appearance of characters. Correlation Registration Mode is based on RSI Search technology, whereas Standard Registration Mode is based on PatMax technology, and the two modes generally reflect the differences between those two technologies.

Which of these modes is best for a given application may need to be determined by experimentation. Some of the trade-offs you may expect are listed hereinafter.

Correlation Registration Mode advantages:

- It can be faster, especially when DOF search ranges are small.
- It can handle smaller characters.
- It can handle somewhat blurrier characters.
- It can handle some types of image noise and texture better.

Correlation Registration Mode disadvantages:

- Tools can use up much more memory, especially when non-translational DOF ranges are large in order to store multiple templates.
- It can be slower for large DOF ranges.
- It cannot handle nonlinear brightness changes.

If you need a relatively small DOF range and are verifying print on an ordinary printing surface such as a package or label, Correlation Registration Mode may be the best choice. For printing on backgrounds with other features or text stamped into metal or other such surfaces, Standard Registration Mode will usually be the best choice.

OCVMax Optimization

Cognex suggests the following for the OCVMax tool:

- Use a fixture, such as a PatMax pattern trained on image features that have a consistent positional relationship to the text, and the search mode Position + Shift Uncertainty.

  By using a fixture, your vision application can automatically compensate for the location of the symbols and paragraphs in successive run-time images.

  The search mode Position + Shift Uncertainty starts a search for each paragraph based upon where the paragraph was found in the training image.

  If your images do not contain a suitable candidate for a fixture and the position of the character strings can vary significantly between images, specify a larger value for *X Shift* and *Y Shift* than the default values returned by the tuning feature. If the OCVMax tool still cannot reliably locate the paragraph, switch to a different search mode.

- Enable and use character key search parameters when the character keys in your paragraphs appear on a rotated surface.

  Characters that appear on a rotated surface often exhibit changes in angle, position, and shear when compared to one another.

- Choose an appropriate level of image noise when you train your OCVMax tool if you use the Standard Registration Mode.

  If the run-time images contain a high degree of irrelevant image data, the OCVMax tool can likely generate low matching scores which decreases the chance the tool will correctly verify a character string even when the string contains the correct characters.

- Enable the shear degree of freedom, in both image search parameters as well as character key search parameters, as recommended by the OCVMax tool when you test the tool against the sample images your vision application is likely to acquire.

  By default, the OCVMax tool tuning feature does not enable a factor for shear as it determines good starting values for various degrees of freedom, as enabling shear usually increases the time required by the tool to analyze an image.

- As you specify the font an OCVMax tool will use, you enable which characters in the font the tool will consider. Use separate OCVMax tools for constant paragraphs where the potential for confusion between mutually exclusive characters can be high.

  For example, if one paragraph contains the letter "O" and another contains the letter "Q", use separate OCVMax tools and enable one letter while excluding the other from consideration.

- The tuning feature can require a significant amount of time depending on the type and number of paragraphs you want to verify.

  In general, an OCVMax tool will complete the tuning process faster with separate OCVMax tools of one paragraph than a single OCVMax tool with multiple paragraphs.

Differences from the OCV Tool

The OCVMax tool described in this chapter differs from the OCV tool described in the chapter *OCV Tool* on page 496 in the following ways:

- Image-based training:

  For every character your vision application must verify, the OCV tool requires you to generate a font model sampled from one or more acquired images. The OCVMax tool uses an existing font file to define the model for

each character, but the font file can be generated from an image or produced synthetically. If you have the font file for the characters in the strings you want to verify, configuring an OCVMax tool is generally quicker than configuring an OCV tool.

- Search reliability:

  The OCVMax tool uses PatMax technology to search for each character string, allowing greater reliability in locating each string as its position varies in successive images.

- Distortion:

  The OCVMax tool can handle far more image-to-image character distortion than the OCV tool.