ALACRON

FASTSERIES

USER'S MANUAL MACHINE VISION LIBRARY (MviL)

30002-00269

Document Name:	ALRT RT SW Programmer's Guide & Reference User's Manual	
Document Number:	30002-00169	
Revision History:	1.2	June 13, 2003
	1.3	Oct 23, 2003

Trademarks:

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
Fax: 603-891-2745

Web Site:
http://www.alacron.com/

Email:
sales@alacron.com, or support@alacron.com

Table of Contents

1 INTRODUCTION

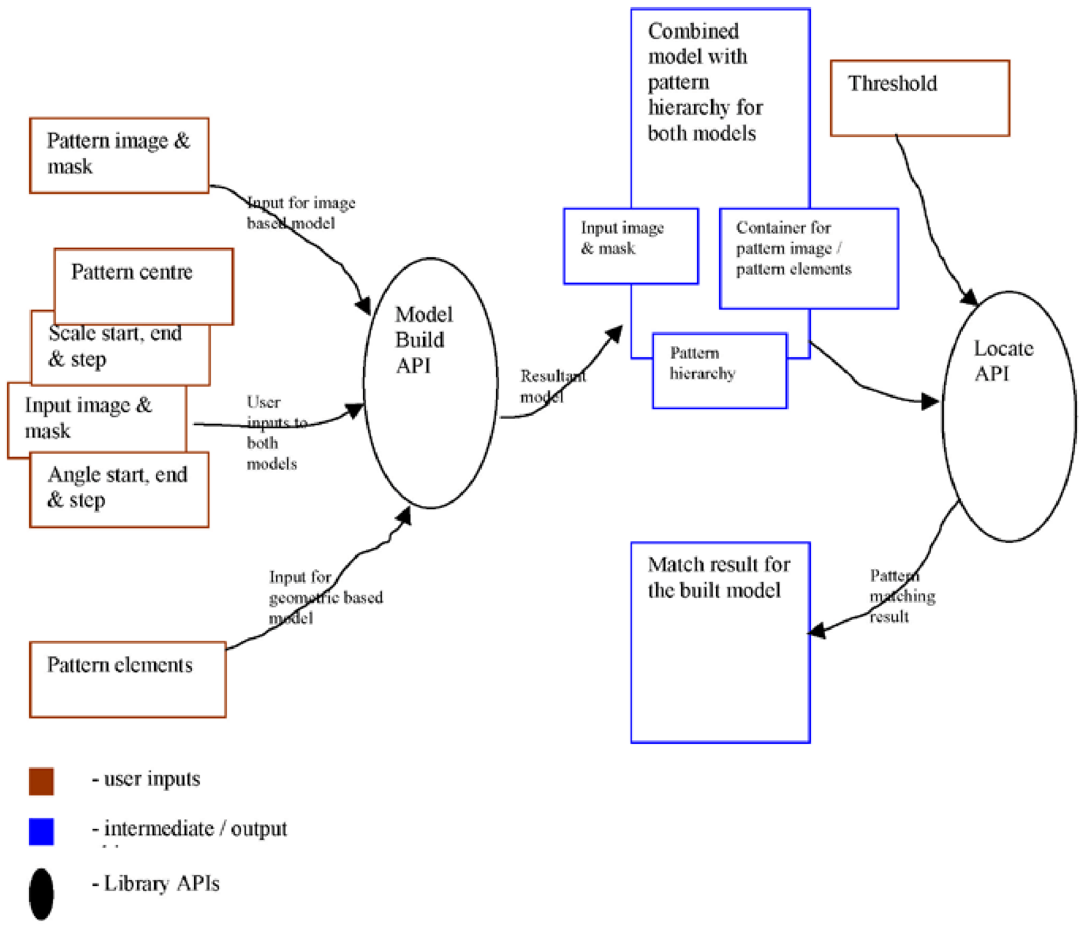
In this document the APIs built for Pattern recognition, Calibration/Measurement subsystem, Morphology and BLOB analysis within the MViL library are described.

This section will elicit the build procedure and code structuring of the various library components in MVIL. MVIL library is constituted of 3 components – Pattern Recognition, Calibration & Measurement and Morphology & BLOB Analysis. These are provided as template based api's structured in 3 files - MVIL_PatMatch.hpp, MVIL_Calibration.hpp and MVIL_Morph_Blob.hpp. If the user need to use the MVIL template based library api's in his applicaton he will have to include the MVIL.hpp file after FOIL.hpp in his source file. As MVIL is an extension to FOIL the necessary setting for FOIL is to be done. All the MVIL api's expect the Images in the FOIL / MVIL defined Image representation. So user will have to do a preprocessing of reading image files of supported formats and populate that data into the FOIL / MVIL image data structures. Most of the MVIL api's expect the output of some other api execution results as inputs. These are mentioned in arguments section across each api description with example. For more detailed usage of these library components in real world scenario plese refer the last section in this document.

2 MVIL LIBRARY: PATTERN RECOGNITION

2.1 Control/Data Flow Diagrams

Shape Based Pattern Matching



This section details the API's built for Pattern Recognition within the MViL library. In the description of each api the necessary input output components are mentioned

2.2 Pattern Recognition API's

1. FOIL::ModelBuild()
2. FOIL::Locate()

Class:	**CImage Class**
Description:	This is a template-based class composing the basic Image abstraction classes in FOIL. This class will hold the input image, patterns and mask image that define the ROI.

Syntax:	template < class imageClass, int imgType >
	class CImage
	{
	  private:

Class:	**CImage Class**
	```
	int            imageType;
	float          imageAngle;
	float          imageScale;

	public:
	    imageClass MatrixImage;
	    CImage()
	    {
	      imageType    =  imgType;
	    }
	    int getImageType (void)
	    {
	      return imageType;
	    }
	    void setImageAngle ( float angle )
	    {
	     imageAngle = angle;
	    }
	    float getImageAngle( void )
	    {
	      return imageAngle;
	    }
	    void setImageScale( float scale )
	    {
	      imageScale = scale;
	    }
	    float getImageScale( void )
	    {
	      return imageScale;
	    }
	}
	```

Class:	**CImage Class**
Template Parameters	ImageClass: Input should be of CImage class type e.g.: CMatrix8, CGrayImage8, CGrayImage16, etc. ImgTyp: Input should be of integer type e.g.: BLACKWHITE, GRAY8, etc.
Member Functions:	**getImageType**: This will return the image type stores as part of instantiation. **setImageAngle**: This will set the angle of rotation of the stored image. **getImageAngle**: This will get the angle of rotation of the stored image. **setImageScale**: This will set the scale of rotation of the stored image. **getImageScale**: This will get the scale of rotation of the stored image.

Class:	**CColourImage Class**
Description:	This is a template-based class that composes the basic Image abstraction classes in FOIL for colour images.

Syntax:	
	```
	template < class imageClass, int imgType>
	class CColourImage
	{
	        private:
	                int             imageType;
	                float            imageAngle;
	                float            imageScale;
	          public:
	                  imageClass* MatrixImage;
	                CColourImage()
	                {
	                  imageType     =  imgType;
	                 MatrixImage = new imageClass(10,10);
	                }
	               int getImageType (void)
	                {
	                  return imageType;
	                }
	                void setImageAngle ( float angle )
	                {
	                        imageAngle = angle;
	                }
	                float getImageAngle( void )
	                {
	                        return imageAngle;
	                }
	                void setImageScale( float scale )
	                {
	```

Class:	**CColourImage Class**

	imageScale = scale;
	}
	}

Template Parameters	ImageClass: Input should be of CImage class type e.g.: CRGBPlanar8, CRGBPlanar16, etc. ImgTyp: Input should be of integer type e.g.: RGBPLANAR8, RGBPLANAR16, etc.
Member Functions:	**getImageType**: This will return the image type stores as part of instantiation. **setImageAngle**: This will set the angle of rotation of the stored image. **getImageAngle**: This will get the angle of rotation of the stored image. **setImageScale**: This will set the scale of rotation of the stored image. **getImageScale**: This will get the scale of rotation of the stored image.

Class:	**CImageModel Class**
Description:	This class will hold the basic image model that will represent the problem space. While building the model, the object of this class will get populated with the problem space.

Syntax:	template <class imageClass, int imgTyp>
	class CImageModel
	{
	private:
	int patternCount;
	int Invariance;
	public:
	CImage <imageClass, imgTyp>InputImage;
	CImage <imageClass, imgTyp>inputImageMask;
	CImage <imageClass, imgTyp>patternMask;
	std::vector<CImage<imageClass, imgTyp>> PatternList;
	
	INDEX_HIERARCHY rotationIndex;
	INDEX_HIERARCHY scaleIndex;
	void incrPatternCount (void)

Class:	**CImageModel Class**

	`{`
	`    patternCount ++;`
	`}`
	`int getPatternCount (void)`
	`{`
	`    return patternCount;`
	`}`
	`void setInvariance (int type)`
	`{`
	`    Invariance = type;`
	`}`
	`int getInvariance (void)`
	`{`
	`return Invariance;`
	`}`
	`}`

Template Parameters	ImageClass: Input should be of CImage class type. e.g.: CGrayImage8, CGrayImage16, CRGBPlanar8, CRGBPlanar16, etc. ImgTyp: Input should be of integer type. e.g. : GRAY8, GRAY16, RGBPLANAR8, RGBPLANAR16, etc.
Member Functions:	**incrPatternCount**: This will increment the pattern count mainitained in the class. **getPatternCount**: This will get the pattern count set in the class. **setInvariance**: This specifies the type of invariance supported in the current model. **getInvariance**: This will get the type of invariance supported in the current model.

Class:	**CGeometricModel Class**
Description:	This class will hold both the image-based model and geometric-based model. We could represent the same problem in both these models and apply matching logic to them.

Class:	**CGeometricModel Class**

Syntax:	template < class imageClass, int imgTyp, class pointType >
	class CGeometricModel
	{
	  private :
	    int patternCount;
	    int basePatternCount;
	  public :
	    CImage < imageClass, imgTyp > InputImage;
	    CImage < imageClass, imgTyp > InputImageMask;
	    std::vector <CPatternElement < imageClass, imgTyp, pointType > * > PatternList;
	    INDEX_HIERARCHY* rotationIndex;
	    INDEX_HIERARCHY* scaleIndex;
	    CGeometricModel()
	    {
	      patternCount = 0;
	      basePatternCount = 0;
	    }
	
	     void incrPatternCount (void)
	     {
	         patternCount ++;
	     }
	     int getPatternCount (void)
	     {
	         return patternCount;
	     }
	     int getBasePatternCount (void)
	     {
	         return basePatternCount;
	     }

Class:	**CGeometricModel Class**

	void setBasePatternCount (int count)
	{
	basePatternCount = count;
	}
	}

Template Parameters	ImageClass: Input should be of CImage class type. e.g.: CGrayImage8, CGrayImage16, CRGBPlanar8, CRGBPlanar16, etc. ImgTyp : Input should be of integer type. e.g.: GRAY8, GRAY16, RGBPLANAR8, RGBPLANAR16, etc. PointType: The data type representing the point coordinates. This is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**incrPatternCount**: This will increment the pattern count mainitained in the class. **getPatternCount**: This will get the pattern count set in the class. **getBasePatternCount**: This will get the base pattern count stored in the class. **setBasePatternCount**: This will set the base pattern count stored in the class.

Class:	**CCombinedModel Class**
Description:	This class will hold both the image-based model and geometric-based model. We could represent the same problem in both these models and apply matching logic to them.

Syntax:	template < class imageClass, int imgTyp, class pointType >
	class CCombinedModel
	{
	private:
	model_type definedModel;
	int invariance;
	float scaleStart;
	float scaleStep;
	float scaleEnd;

Class:	**CCombinedModel Class**
	```
	            float angleStart;
	            float angleStep;
	            float angleEnd;
	            int rotatePatternCount;
	             int scalePatternCount;
	  public:
	             CImageModel < imageClass, imgTyp > Image;
	              CGeometricModel <  imageClass, imgTyp, pointType >
	Geometric;
	              int getDefinedModel(void)
	            {
	                  return definedModel;
	            }
	           void setDefinedModel (model_type modelType)
	           {
	                definedModel = modelType;
	           }
	          float getScaleStart( void )
	          {
	                  return scaleStart;
	          }
	          float getScaleStep( void )
	          {
	                  return scaleStep;
	          }
	          float getScaleEnd( void )
	          {
	                  return scaleEnd;
	          }
	          void setScaleStart( float start )
	          {
	```

Class:	CCombinedModel Class

	scaleStart = start;
	}
	void setScaleStep(float step)
	{
	scaleStep = step;
	}
	void setScaleEnd(float end)
	{
	scaleEnd = end;
	}
	float getAngleStart(void)
	{
	return angleStart;
	}
	float getAngleStep(void)
	{
	return angleStep;
	}
	float getAngleEnd(void)
	{
	return angleEnd;
	}
	void setAngleStart(float start)
	{
	angleStart = start;
	}
	void setAngleStep(float step)
	{
	angleStep = step;
	}
	void setAngleEnd(float end)

Class:	**CCombinedModel Class**
	```
	        {
	            angleEnd = end;
	        }
	        void setInvariance ( int type )
	        {
	            invariance = type;
	        }
	        int getInvariance ( void )
	        {
	            return invariance;
	        }
	        void setRotatePatternCount ( int rCount )
	        {
	            rotatePatternCount = rCount;
	        }
	        int getRotatePatternCount ( void )
	        {
	            return rotatePatternCount;
	        }
	        void setScalePatternCount ( int sCount )
	        {
	            scalePatternCount = sCount;
	        }
	            int getScalePatternCount ( void )
	        {
	            return scalePatternCount;
	        }
	}
	```
Template Parameters	ImageClass: Input should be of CImage class type. e.g.: CGrayImage8, CGrayImage16, CRGBPlanar8, CRGBPlanar16, etc. ImgTyp: Input should be of integer type. e.g.: GRAY8, GRAY16, RGBPLANAR8, RGBPLANAR16, etc.

Class:	**CCombinedModel Class**
	PointType: The data type representing the point coordinates. This is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**getDefinedModel**: This will get the types of model built within the common container. **setDefinedModel**: This will set the type of model built within the common container. **getScaleStart**: This will get the start scale factor stored in the class. **setScaleStart**: This will set the start scale factor in the class. **getScaleStep**: This will get the scale step factor stored in the class. **getScaleStep**: This will set the scale step factor in the class. **getScaleEnd**: This will get the end scale factor stored in the class. **setScaleEnd**: This will set the end scale factor in the class. **getAngleStart**: This will get the start angle factor stored in the class. **setAngleStart**: This will set the start angle factor in the class. **getAngleStep**: This will get the angle step factor stored in the class. **getAngleStep**: This will set the angle step factor in the class. **getAngleEnd**: This will get the end angle factor stored in the class. **setAngleEnd**: This will set the end angle factor in the class. **setInvariance**: This specifies what type of invariance is supported in the current model. **getInvariance**: This will get what type of invariance is supported in the current model. **setRotatePatternCount**: This sets the number of rotated patterns stored in the hierarchy based on the angle factors. **getRotatePatternCount**: This gets the number of rotated patterns stored in the hierarchy based on the angle factors. **setScalePatternCount**: This sets the number of scaled patterns stored in the hierarchy based on the scale factors. **getScalePatternCount**: This gets the number of scaled patterns stored in the hierarchy based on the scale factors.

Class:	**CPoint Class**

Class:	**CPoint Class**
Description:	This is a template-based class defining the basic point object.

Syntax:	template <class pointType>
	class CPoint
	{
	    public:
	        pointType x;
	        pointType y;
	}

Template Parameters	pointType: Input should be of the data type of image's pixel. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Summary:	**Builds the problem space for pattern matching**

Syntax:	
	```
	err_ret FOIL::ModelBuild (
	        model_type                          modelType,
	        CImage<imageClass, imgTyp>          inputImage,
	        CImage<imageClass, imgTyp>          pattern,
	        CImage<imageClass, imgTyp>
	    inputImageMask,
	        CImage<imageClass, imgTyp>          patternMask,
	        float                               scaleStart,
	        float                               scaleStep,
	        float                               scaleEnd,
	        float                               angleStart,
	        float                               angleStep,
	        float                               angleEnd,
	        CCombinedModel<imageClass,imgTyp,pointType>
	&resultModel,
	        CPoint <unsigned long>              patternCentre
	);
	```

Arguments:	modelType – Parameter to indicate the kind of problem space being built. This parameter can assume the valid values MODELIMAGECORRELATION and MODELIMAGEDIFFERENCE, which are enumerated FOIL types. inputImage – Input image on which the pattern image is to be recognized. This should be a non-null image object. pattern – Pattern image to be searched for on the input image. This should be a non-null image.

Summary:	**Builds the problem space for pattern matching**
	inputImageMask – The ROI mask to be applied on the input image. The pixel locations where the user does not want to include in computation, should hold ‘0’ value. All non-zero values. patternMask – The ROI mask to be applied on the pattern image. The pixel locations where the user does not want to include in computation, should hold ‘0’ value. All non-zero values. scaleStart – Starting scale for the pattern. The valid range will start from 1 till a scale value which, when applied on the pattern, will be less than the input dimensions. scaleStep – Step to compute next successive scale factor from scaleStart. scaleEnd – Specifies the end scale. The valid range is a non-zero value. Ideally it should be greater than scaleStart. angleStart – Starting angle for the pattern. Values range from 0 to 360 degrees. angleStep – Step to compute next successive angle from angleStart. It should be a non-zero number angleEnd – Specifies the end angle. Values range from 0 to 360 degrees. But ideally should be greater than angleStart. resultModel – Object containing the resultant model. This should be non-null object. patternCentre – Pattern center point provided by user. This should be a non-null object with x coordinate between the pattern row boundary and y coordinate between pattern col boundary.
Return:	API execution status.
Description:	This API constructs the problem space for image-based correlation, image-based difference, geometric-based pattern matching, and all combinations of the three pattern-matching problems. The input and pattern image and their ROI masks are manipulated with the scale and rotational invariance parameters to construct the problem space. When used to construct geometric-based pattern-matching problem space, a pre-processing step is required. After instantiating the resultModel object, add the base point patterns to the PatternList vector (*Geometric member* object) before invoking ModelBuild.

Example:	FOIL::CImage<FOIL::CGrayImage8, GRAY8> inputImage;
	FOIL::CImage<FOIL::CGrayImage8, GRAY8> inputImageMask;
	FOIL::CImage<FOIL::CGrayImage8, GRAY8> patternImage;
	FOIL::CImage<FOIL::CGrayImage8, GRAY8> patternMask;
	FOIL::CCombinedModel<FOIL::CGrayImage8,GRAY8,unsigned long> resultModel;
	FOIL::CPoint<unsigned long> patternCentre;
	FOIL CPatternElement<FOIL::CGrayImage8, GRAY8,int> basePattern1;
	FOIL:: CPatternElement<FOIL::CGrayImage8, GRAY8,int> basePattern2;
	FOIL:: CPatternElement<FOIL::CGrayImage8, GRAY8,int> basePattern3;

Summary:	**Builds the problem space for pattern matching**

	float scaleStart;
	float scaleStep;
	float scaleEnd;
	float angleStart;
	float angleStep;
	float angleEnd;
	int api_status = -1;
	
	// FOR IMAGE-BASED
	api_status = FOIL::ModelBuild (FOIL::MODELIMAGECORRELATION, inputImage, patternImage, inputImageMask, patternMask, scaleStart, scaleStep, scaleEnd, angleStart, angleStep, angleEnd, resultModel, patternCentre);
	
	// FOR GEOMETRIC-BASED
	/* Pre-processing step */
	resultModel.Geometric.PatternList.push_back(basePattern1);
	resultModel.Geometric.PatternList.push_back(basePattern2);
	resultModel.Geometric.PatternList.push_back(basePattern3);

Class:	**CPatternElement Class**
Description:	This is a template-based class defining the basic pattern element object to be searched. We will have to derive various classes from this base class to override the POP function to provide either picking a single pixel value or applying the sobel operation.

Syntax:	template < class imageClass, int imgTyp, class pointType >
	class CPatternElement
	{
	public:
	CImage < imageClass, imgTyp > PatternElement;
	float expected;
	float weight;
	CPoint < pointType > Point;
	virtual float PoP(CImage < imageClass, imgTyp > element, CPoint < pointType > p)
	{
	return (element.MatrixImage.GetItem(p.x, p.y));

	}
	 }

Template Parameters	ImageClass :Input should be of CImage class type e.g.: CGrayImage8, CGrayImage16, CRGBPlanar8, CRGBPlanar16, etc. ImgTyp : Input should be of integer type. e.g. : GRAY8, GRAY16, RGBPLANAR8, RGBPLANAR16 etc PointType : The data type representing the point coordinates. This is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**POP**: This is the overloaded api defining the point operator for the pattern element.

Class:	**CPatternPointSobel Class**
Description:	This is a template-based class defining the basic pattern element object to be searched. We will have to derive various classes from this base class to override the POP function to provide either picking a single pixel value or applying the sobel operation.

Syntax:	template < class imageClass, int imgTyp, class pointType >
	 class FOIL_API CPatternPointSobel : public CPatternElement < imageClass, imgTyp, pointType >
	 {
	 public :
	 float PoP(CImage < imageClass, imgTyp > element, CPoint < unsigned long > p);
	 }

Template Parameters	ImageClass :-Input should be of CImage class type e.g.: CGrayImage8, CGrayImage16, CRGBPlanar8, CRGBPlanar16 etc ImgTyp: Input should be of integer type e.g.: GRAY8, GRAY16, RGBPLANAR8, RGBPLANAR16 etc PointType: The data type representing the point coordinates. This is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc
Member Functions:	**POP**: This is the overloaded api defining the point operator for the pattern element. Here we use Sobel to get the result of point operation

Class:	**CMatch Class**
Description:	This is a template-based class that will hold the match results of a pattern matching operation.

Syntax:	template <class pointType>
	class CMatch
	{
	    public:
	        float score;
	        CPoint <pointType> point;
	}

Template Parameters	pointType: Input should be of the data type of image's pixel e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Class:	**CMatchList Class**
Description:	This class will hold the result of the matches and the resultant image depicting the match.

Syntax:	template < class pointType >
	class FOIL_API CMatchList
	{
	    private :
	        int matchCount;
	    public :
	        float angle;
	        float scale;
	        std::vector < CMatch < pointType > > MatchList;
	        CMatchList()
	        {
	            matchCount = 0;
	        }
	        void incrMatchCount (void)
	        {

Class:	**CMatchList Class**

	matchCount++;
	}
	int getMatchCount (void)
	{
	return matchCount;
	}
	}

Template Parameters	pointType:Input should be of the data type of image's pixel e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**incrMatchCount**: This will increment the match count mainitained in the class. **getMatchCount**: This will get the match count set in the class

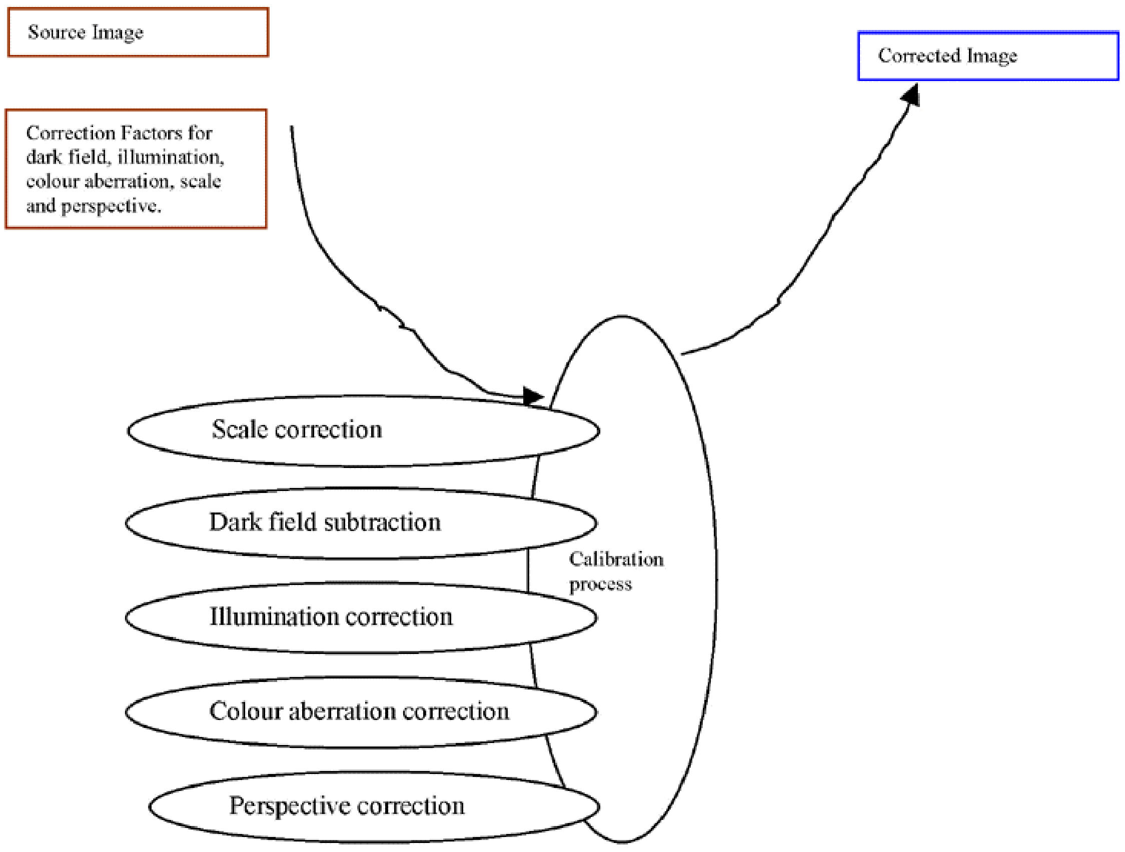
Summary:	**Locates a pattern on a pre-built model**
Syntax:	err_ret FOIL::Locate (model_type modelType, float threshold, CCombinedModel <imageClass,imgTyp,pointType> &inputModel, CMatchList <centreType> &outputMatchList);
Arguments:	modelType - Indicates input image model on which the pattern is to be located. This can take the valid ranges of FOIL enumerated types among MODELIMAGECORRELATION, MODELIMAGEDIFFERENCE and MODELGEOMETRIC threshold – Value to compare computed scores to determine a match. For the correlation approach, the threshold has a valid range between 0.0 and 1.0. For the other 2 approaches it is governed by the input specified by the user. inputModel – Problem space populated by ModelBuild. This is a non-null object populated by the ModelBuild function for Correlation or Difference based matching. In Geometric based matching, the process of adding base pattern points is done prior to this call. outputMatchList – List of matched pixel coordinates and computed scores. This should be a non-null object.
Return:	API execution status.
Description:	This API locates a pattern on an image-based correlation, image-based difference, or geometric-based pattern-matching problem space.
Example:	FOIL::CMatchList<unsigned long> match;

	float threshold = 1.0;
	
	api_status = FOIL::Locate(FOIL::MODELIMAGECORRELATION, threshold, resultModel, match);

3 MVIL LIBRARY: CALIBRATION AND MEASUREMENT

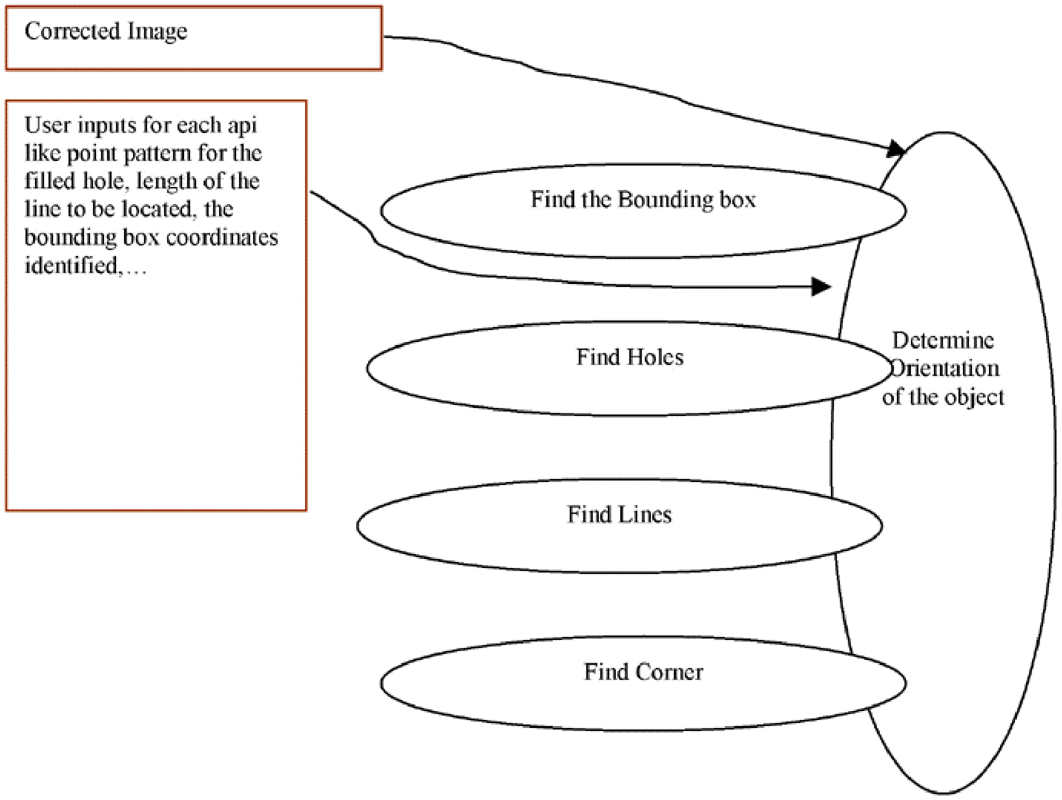
3.1 Control/Data Flow Diagrams

Calibration

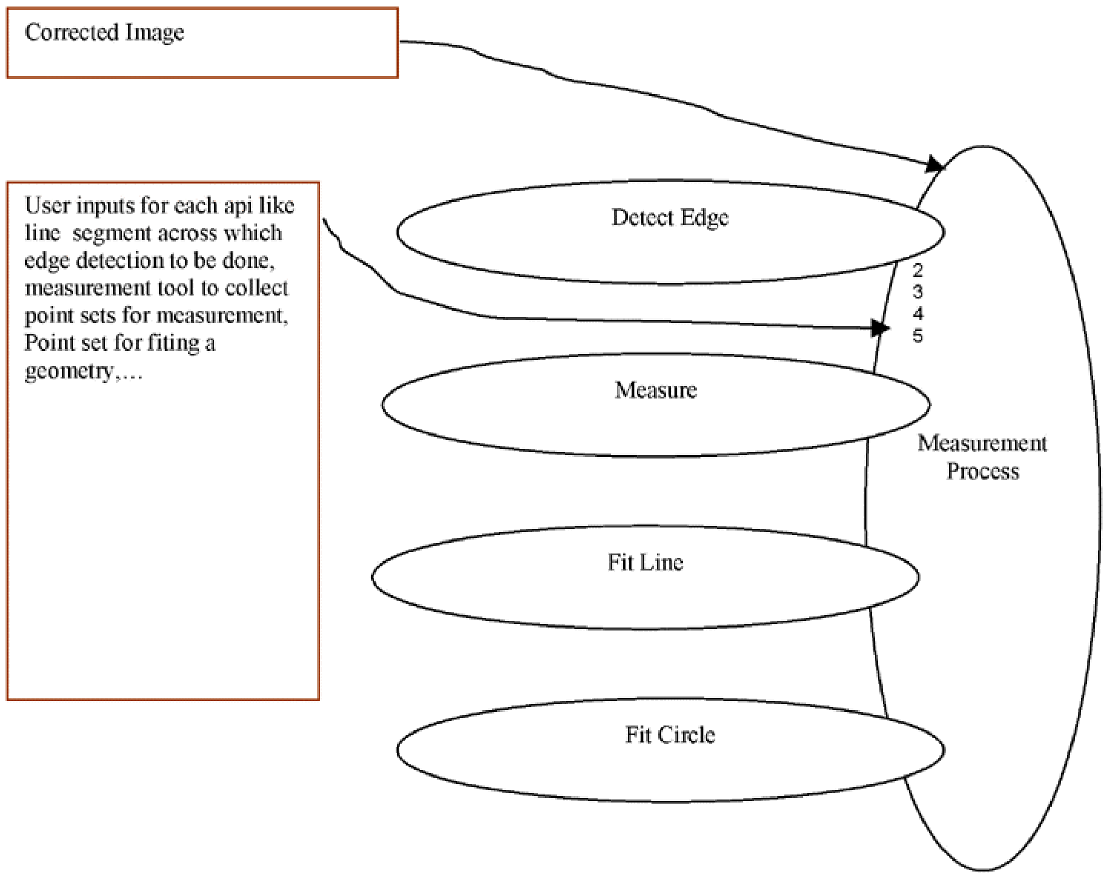


●	- Library apis
🟥	- user inputs
🟦	- intermediate / outputs

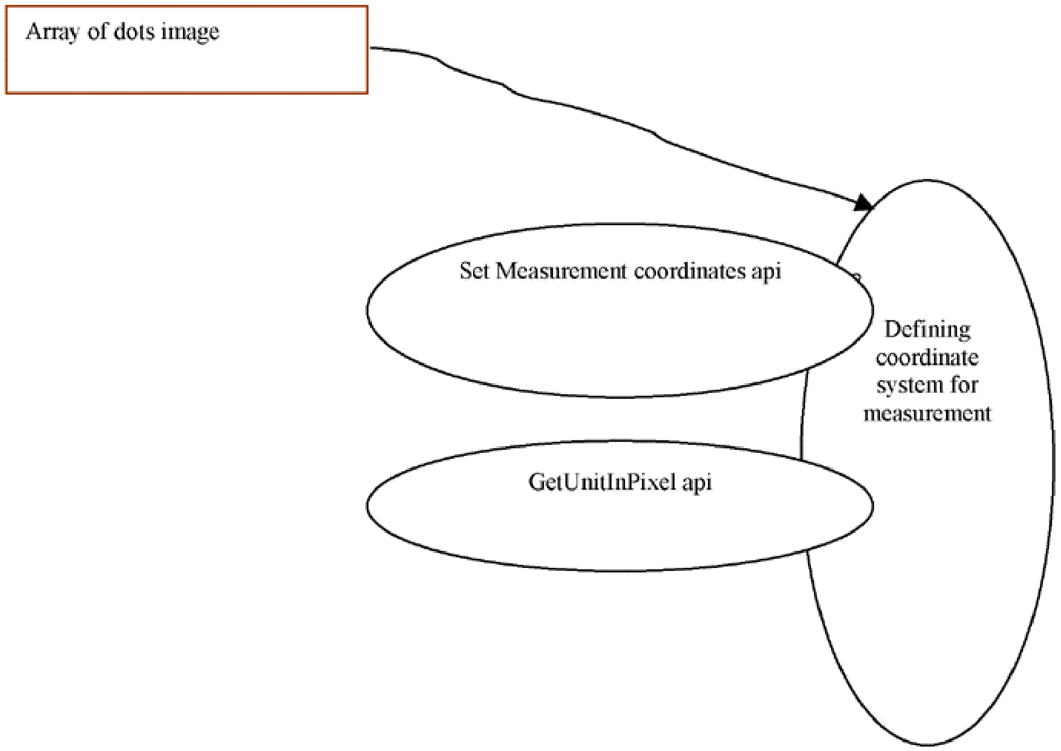
Orientation


Corrected Image
User inputs for each api like point pattern for the filled hole, length of the line to be located, the bounding box coordinates identified,…
Find the Bounding box
Find Holes
Find Lines
Find Corner
Determine Orientation of the object

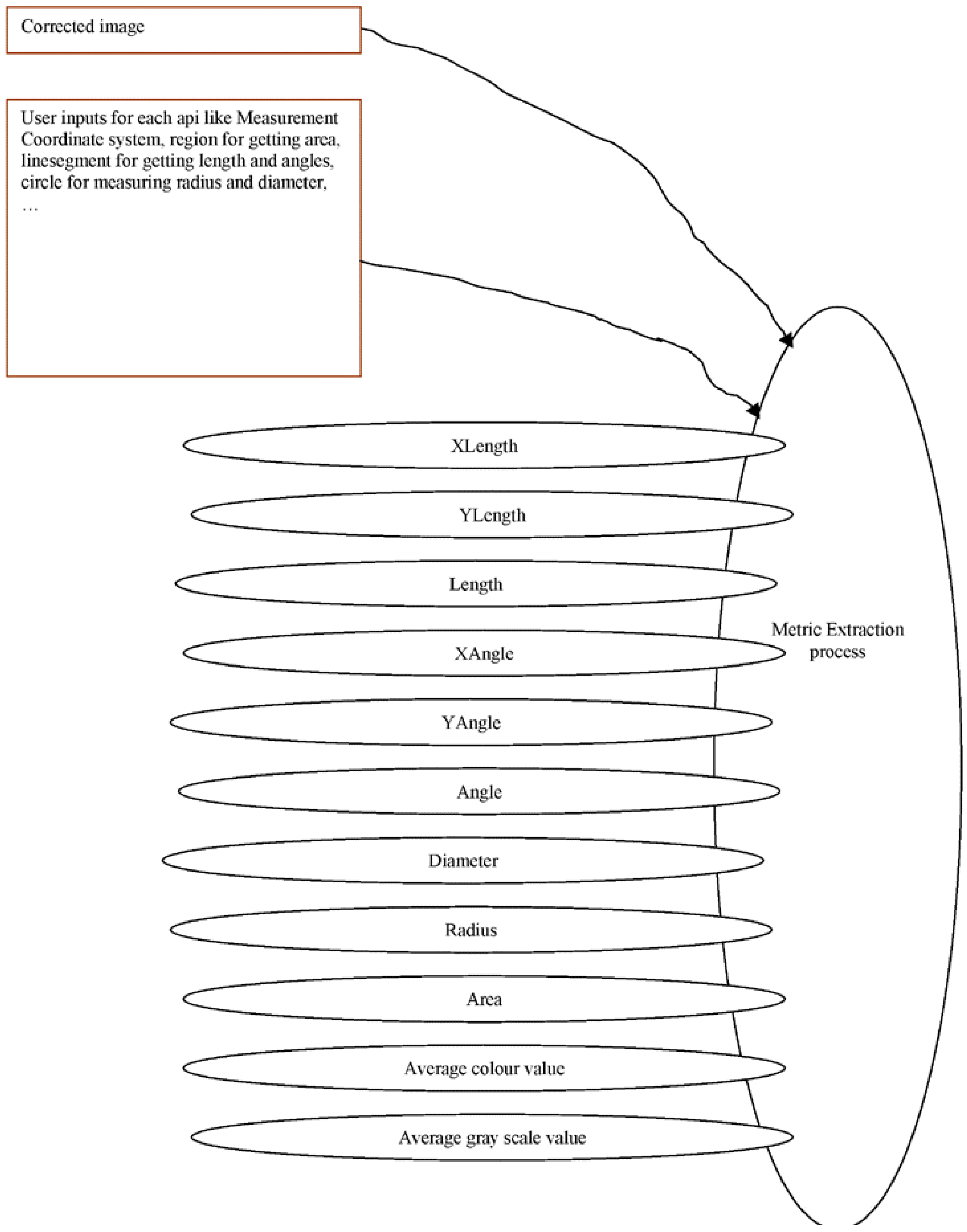
Measurement


Corrected Image
User inputs for each api like line segment across which edge detection to be done, measurement tool to collect point sets for measurement, Point set for fiting a geometry,…
Detect Edge
2
3
4
5
Measure
Fit Line
Fit Circle
Measurement Process

Coordinate Definition


Array of dots image
Set Measurement coordinates api
GetUnitInPixel api
Defining coordinate system for measurement

Metric Extraction


Corrected image
User inputs for each api like Measurement Coordinate system, region for getting area, linesegment for getting length and angles, circle for measuring radius and diameter, …
XLength
YLength
Length
XAngle
YAngle
Angle
Diameter
Radius
Area
Average colour value
Average gray scale value
Metric Extraction process

This section details the API's built for Calibration and Measurement in the MViL library. In the description of each API, the necessary input output components are mentioned.

Subcategories of APIS included in this section are:

- Calibration

- Measurement Coordinate System
- Orientation
- Measurement and Fit
- Spatial Metric Tool
- Photometric Tool

3.2 Required Classes for Calibration APIs

Class:	**CCalibratedImage Class**
Description:	This template based class will expose the various correction API's to be applied on the input image and template image.

Syntax:	template < class imageClass, int imgTyp, class dataType>
	class CCalibratedImage
	{
	public:
	static CImage<imageClass , imgTyp> **DarkFieldCorrection** (CImage<imageClass , imgTyp> inputImage, CImage<CMatrixF,FLOAT> darkFeild, float** gainFactor);
	static CColourImage<imageClass, imgTyp> **DarkFieldCorrection** (CColourImage<imageClass,imgTyp> inputImage, CImage<CMatrixF,FLOAT> rChannelDarkFeild, CImage<CMatrixF,FLOAT> gChannelDarkFeild, CImage<CMatrixF,FLOAT> bChannelDarkFeild, float** rChannelGainFactor, float** gChannelGainFactor, float** bChannelGainFactor);
	static CImage<imageClass , imgTyp> **PerspectiveCorrection** (CImage<imageClass , imgTyp> inputImage, CVector<long> u_vector, CVector<long> v_vector, CVector<long> n_vector, float L);
	static CColourImage<imageClass,imgTyp> **ColourAberrationCorrection** (CColourImage<imageClass,imgTyp> inputImage, double scaleFactorRed, double scaleFactorGreen, double scaleFactorBlue, double scaleCentreX, double scaleCentreY);
	static CImage<imageClass , imgTyp> **ScaleCorrection** (CImage<imageClass, imgTyp> inputImage, float ScaleX, float ScaleY, float ShearX, float ShearY, float TransX, float TransY);
	}

Template Parameters	ImageClass:Input should be of CImage class type e.g.: CMatrix8, CGrayImage8, CGrayImage16, etc.

	ImgTyp: Input should be of integer type e.g.: BLACKWHITE, GRAY8, etc. DataType: The data type representing the pixel data type of the image. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**DarkFieldCorrection**: This overloaded function will perform the dark field subtraction based on the correction factor entered by the user. In case of colour images, the function taking correction factors is applied for each channel. This api will return the corrected image object of type Cimage. **ColourAberrationCorrection**: Performs the colour aberration correction based on the correction factor entered by the user. This api will return the corrected image object of type Cimage. **ScaleCorrection**: Performs the scale correction on the input image. If scale factor is > 1, the algorithm will have to fill the additional pixel with bilinear interpolation. **Perspective Correction:** Performs the perspective correction on a grey scale input image per the entries made by the user.

3.3 Calibration API's

This section details the API's built for Measurement subsystem within the MViL library.

1. CCalibratedImage::DarkFieldCorrection()
2. CCalibratedImage::ColourAberrationCorrection()
3. CCalibratedImage::ScaleCorrection()
4. CCalibratedImage::PerspectiveCorrection()

Summary:	**Performs dark field subtraction on the input image for gray scale type images**
Syntax:	CImage<imageClass, imgTyp> CCalibratedImage::DarkFieldCorrection (CImage<imageClass , imgTyp> inputImage, CImage<CMatrixF,FLOAT> darkFeild, float** gainFactor);
Arguments:	InputImage – Source image on which the correction is to be performed. The input image should be a valid type. It should not be empty. darkFeild – The dark field correction factor.It should be of CImage type with image CMatrixF and image type FLOAT. The valid range is determined by the user requirement. gainFactor – The gain factor. It is a two-dimensional array with a value between 0 and 1.
Return:	Dark field subtracted gray scale image.
Description:	This API performs dark field subtraction on each pixel value in the input

Summary:	**Performs dark field subtraction on the input image for gray scale type images**
	image according to the following formula. For each pixel (i,j) in the input image. calibratedImage[i,j] = gainFactor[i,j] * (inputImage[i,j] - darkField [i,j])

Example:	FOIL::CImage<CGrayImage8, GRAY8> outputImage;
	FOIL::CImage<CGrayImage8, GRAY8> inputImage;
	FOIL::CImage<CMatrixF, FLOAT> correctionFactor;
	float** gain;
	FOIL::CCalibratedImage< CGrayImgae8, GRAY8,float> calibrationObj;
	outputImage = calibrationObj . DarkFieldCorrection(inputImage, correctionFactor,gain);

Summary:	**Performs dark field subtraction on the input image for color type images**
Syntax:	CColourImage<imageClass, imgTyp> CCalibratedImage::DarkFieldCorrection (CColourImage<imageClass,imgTyp> inputImage, CImage<CMatrixF,FLOAT> rChannelDarkFeild, CImage<CMatrixF,FLOAT> gChannelDarkFeild, CImage<CMatrixF,FLOAT> bChannelDarkFeild, float** rChannelGainFactor, float** gChannelGainFactor, float** bChannelGainFactor);
Arguments:	inputImage – Source image on which the correction is to be performed. The input image should be a valid type. It should not be empty. rChannelDarkFeild – The dark field correction factor to be applied on RED channel. It should be of CImage type with image CmatrixF and image type FLOAT. The valid range is governed by user requirements. rChannelGainFactor – The gain factor to be applied to the RED channel. It is a two-dimensional array with a value between 0 and 1. gChannelDarkFeild – The dark field correction factor to be applied on GREEN channel. It should be of CImage type with image CmatrixF and image type FLOAT. The valid range is governed by user requirements. gChannelGainFactor – The gain factor to be applied to the GREEN channel. It is a two-dimensional array with a value between 0 and 1. bChannelDarkFeild – The dark field correction factor to be applied on BLUE channel. It should be of CImage type with image CmatrixF and image type FLOAT. The valid range is governed by user requirements. bChannelGainFactor – The gain factor to be applied to the BLUE channel. It is a two dimensional array with a value between 0 and 1.
Return:	Dark field subtracted color image.
Description:	This overloaded API performs dark field subtraction on the respective

Summary:	**Performs dark field subtraction on the input image for color type images**

	color channels of each pixel value in the input image according to the following formula:
	
	For each pixel (i,j) in the input image:
	REDCHANNEL(calibratedImage[i,j]) = rChannelGainFactor[i,j] * (REDCHANNEL(inputImage[i,j]) - rChannelDarkField [i,j]);
	GREENCHANNEL(calibratedImage[i,j]) = gChannelGainFactor[i,j] * (GREENCHANNEL(inputImage[i,j]) - gChannelDarkField [i,j]);
	BLUECHANNEL(calibratedImage[i,j]) = bChannelGainFactor[i,j] * (BLUECHANNEL(inputImage[i,j]) - bChannelDarkField [i,j]);

Example:	FOIL::CColourImage<CRGBPlanar8, RGBPLANAR8> outputImage;
	FOIL::CColourImage< CRGBPlanar8, RGBPLANAR8> inputImage;
	FOIL::CImage<CMatrixF, FLOAT> RedFactor;
	FOIL::CImage<CMatrixF, FLOAT> GreenFactor;
	FOIL::CImage<CMatrixF, FLOAT> BlueFactor;
	float** redGain;
	float** greenGain;
	float** blueGain;
	FOIL::CCalibratedImage< CRGBPlanar8, RGBPLANAR8, float> calibrationObj;
	
	outputImage = calibrationObj . DarkFieldCorrection(inputImage, RedFactor, GreenFactor, BlueFactor, redGain, greenGain, blueGain);

Summary:	**Performs Color aberration correction on the input image for color type images**
Syntax:	CColourImage<imageClass,imgTyp> CCalibratedImage::ColourAberrationCorrection (CColourImage<imageClass,imgTyp> inputImage, double scaleFactorRed, double scaleFactorGreen, double scaleFactorBlue, double scaleCentreX, double scaleCentreY);
Arguments:	inputImage – Source image on which the correction is to be performed. The input image should be a valid colour type. It should not be empty. rChannelCorrectionFactor – The colour aberration correction factor to be applied on RED channel. It should be of CImage type with image CmatrixF and image type FLOAT. scaleFactorRed – The sale factor to be applied on RED channel. The valid range is governed by user requirements. scaleFactorGreen – The sale factor to be applied on GREEN channel. The valid range is governed by user requirements. scaleFactorBlue – The sale factor to be applied on BLUE channel. The

Summary:	**Performs Color aberration correction on the input image for color type images**
	valid range is governed by user requirements. scaleCentreX – The Xcenter of the scale. The valid range is governed by user requirements. scaleCentreY – The Ycenter of the scale. The valid range is governed by user requirements.
Return:	Color aberration corrected image.
Description:	For each color channel, the algorithm will apply the respective scale factors and compute the bilinear point for the corresponding output pixels.

Example:	FOIL::CColourImage<CRGBPlanar8, RGBPLANAR8> outputImage;
	FOIL::CColourImage< CRGBPlanar8, RGBPLANAR8> inputImage;
	double RedFactor;
	double GreenFactor;
	double BlueFactor;
	double Xcentre;
	double Ycentre;
	FOIL::CCalibratedImage< CRGBPlanar8, RGBPLANAR8, float> calibrationObj;
	
	outputImage = calibrationObj . ColourAberrationCorrection (inputImage, RedFactor, GreenFactor, BlueFactor, Xcentre, Ycentre);

Summary:	**Performs Scale correction on the input image for gray scale type images**
Syntax:	CImage<imageClass , imgTyp> CCalibratedImage::ScaleCorrection (CImage<imageClass, imgTyp> inputImage, float ScaleX, float ScaleY, float ShearX, float ShearY, float TransX, float TransY);
Arguments:	inputImage – Source image on which the correction is to be performed. The input image should be a valid type. It should not be empty. ScaleX – The scale factor on X-axis. This variable should contain a valid float value. The valid range is governed by user requirements. ScaleY – The scale factor on Y-axis. This variable should contain a valid float value. The valid range is governed by user requirements. ShearX – The shear factor on X-axis. This variable should contain a valid float value. The valid range is governed by user requirements. ShearY – The shear factor on Y-axis. This variable should contain a valid float value. The valid range is governed by user requirements. TransX – The translation factor on X-axis. This variable should contain a valid float value. The valid range is governed by user requirements. TransY – The translation factor on Y-axis. This variable should contain a valid float value. The valid range is governed by user requirements.
Return:	Scale-corrected image.
Description:	Input gray scale image is scale corrected. SacleX and ScaleY should be >=

Summary:	**Performs Scale correction on the input image for gray scale type images**
	1.

Example:	FOIL::CImage<CGrayImage8, GRAY8> outputImage;
	FOIL::CImage<CGrayImage8, GRAY8> inputImage;
	float XScale = 2;
	float YScale = 2;
	float XShear = 1;
	float YShear = 1;
	float XTrans = 30;
	float YTrans = 50;
	FOIL::CCalibratedImage< CGrayImage8, GRAY8, float> calibrationObj;
	
	outputImage = calibrationObj . ScaleCorrection (inputImage, XScale, YScale, XShear, YShear, XTrans, YTrans);

Class:	**CVector Class**
Description:	This template-based calss will represent a vector having values across x, y and z directions.

Syntax:	template<class vector_type>
	class CVector
	{
	vector_type x;
	vector_type y;
	vector_type z;
	}

Template Parameters	vector_type:-Input should be of the data type of the vector e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Summary:	**Performs Perspective correction on the input image for gray scale type images**
Syntax:	CImage<imageClass,imgTyp> CCalibratedImage ::PerspectiveCorrection (CImage<imageClass , imgTyp> inputImage, CVector<long> u_vector, CVector<long> v_vector, CVector<long> n_vector, float L);
Arguments:	inputImage – Source image on which the correction is to be performed. The input image should be a valid type. It should not be empty. u_vector – Transformation vector supplied by user. The valid range is governed by user requirements.

Summary:	**Performs Perspective correction on the input image for gray scale type images**
	v_vector – Transformation vector supplied by user. The valid range is governed by user requirements. n_vector – Transformation vector supplied by user. The valid range is governed by user requirements. L – The sensor specification given by the user. The type of sensor used by the user governs the valid range.
Return:	Perspective-corrected image.
Description:	This api will compute the destimation pixel coordinates based on the transformation factors given by the user. Upon boundary checking the sources pixels are copied into the newly computed destination pixel location.

Example:	FOIL::CImage<CGrayImage8, GRAY8> outputImage;
	FOIL::CImage<CGrayImage8, GRAY8> inputImage;
	CVector<unsigned int> u;
	CVector<unsigned int> v;
	CVector<unsigned int> n;
	float L;
	FOIL::CCalibratedImage< CGrayImage8, GRAY8, float> calibrationObj;
	
	outputImage = calibrationObj . PerspectiveCorrection (inputImage, u, v, n, L);

4 REQUIRED CLASSES FOR THE MEASUREMENT COORDINATE SYSTEM:

Class:	**CCoordinateSystem Class**
Description:	This template-based class will represent the generated coordinate system from the array-of-dots image.

Syntax:	template <int rowHoles, class pointTyp, class imageClass, int imgTyp>
	class CCoordinateSystem
	{
	    private:
	        pel_coord Ordinates[rowHoles][rowHoles];
	    public:
	        err_ret SetMeasurementCoordinates
	         (CImage<imageClass,imgTyp> templateImage,

Class:	**CCoordinateSystem Class**

	PatternElement<pointTyp,imgClass,imgTyp> holePattern, float
	threshold);
	int GetUnitInPixels(void);
	}.

Template Parameters	RowHoles:Input should be a Integer value,it will indicate the row count ImageClass:Input should be of Cimage class type e.g.: CMatrix8, CGrayImage8, CGrayImage16, etc. ImgType: Input should be of integer type e.g.: BLACKWHITE, GRAY8, etc.
Member Functions:	**SetMeasurementCoordinates**: This api will get the corrected array-of-dots image and will perform a geomtetric-pattern matching for dots and retruns the user with the two-dimensional coordinate system defined with respect to the array-of-dots image. This coordinate system will have an origin starting with the top left dot in the image. **GetUnitInPixels** – This api will get the generated coordinate system and will return the number of pixels representing one physical unit.

4.1 Measurement Coordinate System API's

1. CCoordinateSystem::SetMeasurementCoordinates()
2. CCoordinateSystem::GetUnitInPixels()

Summary:	**Defines the measurement coordinate system from the array of dots image**
Syntax:	err_ret CCoordinateSystem::SetMeasurementCoordinates(CImage<imageClass,imgTyp> templateImage, CPatternElement<pointTyp,imgClass,imgTyp> holePattern, float threshold);
Arguments:	templateImage: This is the array-of-dots image that is used for defining the coordinate system. This should be a non-null object holePattern: This is the point pattern representation of the filled hole that is to be searched in the template image for defining the coordinate system. This should be a non-null object. Threshold: The threshold input to do the hole location in template image. The input image should be a valid type. It should not be empty. This is governed by the user inputs given for the point pattern representation of a

	hole.
Return:	API execution status.
Description:	A coordinate system is defined from an array-of-dots image by performing a geometric-based pattern matching of a filled hole. This coordinate system allows translation from measurement units to real world units.

Example:	FOIL::CImage<CGrayImage8, GRAY8> arrayOfDots10by10Image;
	FOIL::CImage<CGrayImage8, GRAY8> pointPattern;
	float threshold = 1;
	FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate;
	
	ordinate . SetMeasurementCoordinates(arrayOfDots10by10Image, pointPattern, threshold);

Summary:	**Extracts the pixel distance of array of dots**
Syntax:	int CCoordinateSystem::GetUnitInPixels(void);
Arguments:	None
Return:	Center-to-center pixel distance between two adjacent holes in the array-of-dots image.
Description:	The distance of two adjacent dots in the array of dots image is found within a pre-defined coordinate system.

Example:	FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate;
	int dotLength;
	
	dotLength = ordinate . GetUnitInPixels(void);

4.2 Required Classes for the Orientation APIs

Class:	**COrient Class**
Description:	This template-based class will expose the APIs that will enable the user to get the orientation of the object in the input image

Syntax:	template < class posTyp, class sizeTyp, class imageClass, class imgTyp,
	            class pointTyp>
	class COrient
	        {
	        public:
	        static CRectangle< posTyp , sizeTyp> GetBoundingBox (CImage<imageClass,imgTyp> inputImage);

Class:	**COrient Class**

	static err_ret FindHole(CImage<imageClass,imgTyp> inputImage, CPatternElement<pointTyp,imgClass,imgTyp> holePattern, float threshold, CRectangle<posTyp,sizeTyp> BoundingBox, HolesList & holesInObject);
	
	static err_ret FindLine(Cmage<imageClass,imgTyp> inputImage, long lineLength, CRectangle<posTyp,sizeTyp> boundingBox, CLineSegment<long> & lineSegment);
	
	static err_ret FindCorner(CImage<imageClass,imgTyp> inputImage, CRectangle<posTyp,sizeTyp> boundingBox, CCornerPair<pointTyp> & cornersInObject);
	}

Template Parameters	posTyp:-Input should be of point type can be int,float,long, etc. sizeTyp:Input for the row and column value can be of int,long, etc. ImageClass:Input should be of Cimage class type e.g.: CMatrix8, CGrayImage8, CGrayImage16, etc. ImgType: Input should be of integer type e.g.: BLACKWHITE, GRAY8, etc. PointTyp: Input should be of type int,floatlong, etc. ,this is the input for the point type.
Member Functions:	**GetBoundingBox**: This api will enable the user to identify the smallesl bounding box of the object in the image. **FindHole**: This api will enable the user to locate holes if present in the object. **FindLine:** This api will enables the user to find a line of specified length in an object in the input image. **FindCorner:** This api will enable the user to locate the corners available in the object boundaries.

4.3 Orientation API's

1. COrient::GetBoundingBox()
2. COrient::FindHole()
3. COrient::FindLine()

4. COrient::FindCorner()

Class:	**CRectangle Class**
Description:	This template-based class will represent a rectangle in terms of its bottom-left point and width and height.

Syntax:	template<class posTyp, class sizeTyp >
	CRectangle
	{
	public:
	CPoint <posTyp> pos;
	CSize <sizeTyp> size;
	}

Template Parameters	pos_type: Input should be of the data type of the coordinate locations. It is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc. size_type: Input should be of the data type of the size of the rectangle. It is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Summary:	**Gets the smallest bounding box of the object in an image**
Syntax:	CRectangle< posTyp , sizeTyp> COrient::GetBoundingBox (CImage<imageClass,imgTyp> inputImage);
Arguments:	inputImage: The image which contains the object whose smallest bounding box is to be found. The input image should be a valid type. It should not be empty.
Return:	Bounding box coordinates if successful.
Description:	A single pixel edge detection algorithm is used to identify the image object boundary, from which the rectangular coordinates that enclose the object are defined.

Example:	FOIL::CImage<CGrayImage8, GRAY8> objectImage;
	FOIL::CRectangle<int,int> minBoundingBox;
	FOIL::COrient<int,int, CGrayImage8, GRAY8,int> orientObj;
	
	minBoundingBox = orientObj . GetBoundingBox (objectImage);

Summary:	**Finds a hole within the object in an image**

Summary:	**Finds a hole within the object in an image**
Syntax:	err_ret COrient::FindHole(CImage<imageClass,imgTyp> inputImage , CPatternElement<pointTyp,imgClass,imgTyp> holePattern, float threshold, CRectangle<posTyp,sizeTyp> BoundingBox, HolesList & holesInObject);
Arguments:	inputImage: The image which contains the object on which the hole is to be located. The input image should be a valid type. It should not be empty. holePattern: This is the point pattern representation of the hole that is to be searched in the input image. Threshold: Threshold input for locating holes in the input image. It should be a valid float value. This will be determined by the user input for the point pattern representation of the hole. BoundingBox: The bounding box coordinates where the pattern matching is to be confined. The row and cols size should not be zero for a CRectangle. holesInObject: Collection of located holes in the object. This is an output parameter which should be non-null.
Return:	API execution status.
Description:	A geometric-based pattern location for the hole within the object in the input image is attempted, and the match list is computed.

Example:	FOIL::CImage<CGrayImage8, GRAY8> objectImage;
	FOIL:: CPatternElement<int, CGrayImage8, GRAY8> holePointPattern;
	FOIL::CRectangle<int,int> minBoundingBox;
	float threshold = 1;
	FOIL::HolesList holes;
	FOIL::COrient<int,int, CGrayImage8, GRAY8,int> orientObj;
	FOIL::err_ret return;
	
	return = orientObj . FindHole (objectImage, holePointPattern, minBoundingBox, threshold, holes);

Class:	**CLineSegment Class**
Description:	This template-based class will represent a line segment as a pair of start and end points.

Syntax:	template<class pointTyp>
	CLineSegment
	{
	  public:
	    CPoint< pointTyp > startPoint;
	    CPoint< pointTyp > endPoint;
	}

Template	pointTyp: Input should be of the data type of the coordinate locations. It is

Class:	**CLineSegment Class**
Parameters	dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Summary:	**Finds a line within the object in an image**
Syntax:	err_ret COrient::FindLine(CImage<imageClass,imgTyp> inputImage, long lineLength, CRectangle<posTyp,sizeTyp> boundingBox, CLineSegment<long> & lineSegment);
Arguments:	inputImage: The image which contains the object on which the line of a specified length is to be located. Image should be a valid image. It should not be empty. lineLength: This length of the line to be located in pixel units. It should be a valid positive float value greater than 0. boundingBox: The bounding box coordinates where the line finding is to be confined. Content of this variable, i.e., boundingBox. pos and boundingBox .size should be valid values. boundingBox .size .rows and boundingBox .size.cols should not be 0. lineSegment: The located line segment if finding is successful. This is an output parameter. This should be a non-null object.
Return:	API execution status.
Description:	A straight line of a specific length is searched for within the object in the specified bounding box region.

Example:	FOIL::CImage<CGrayImage8, GRAY8> objectImage;
	FOIL::CRectangle<int,int> minBoundingBox;
	FOIL::CLineSegment<long> line;
	long length = 25;
	FOIL::COrient<int,int, CGrayImage8, GRAY8,int> orientObj;
	FOIL::err_ret return;
	
	return = orientObj . FindLine (objectImage, length, minBoundingBox, line);

Class:	**CCornerPair Class**
Description:	This is a template-based class that will represent a pair of lines that contains a common endpoint that specifies a corner.
Syntax:	template <class pointTyp> CCornerPair {

Class:	**CCornerPair Class**

	public:
	CLineSegment<pointTyp> line1;
	CLineSegment<pointTyp> line2;
	}

Template Parameters	pointTyp: Input should be of the data type of the coordinate locations. It is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Summary:	**Finds a corner pair within the object in an image**
Syntax:	err_ret COrient::FindCorner(CImage<imageClass,imgTyp> inputImage, CRectangle<posTyp,sizeTyp> boundingBox, CCornerPairs & cornersInObject);
Arguments:	inputImage: The image which contains the object on which the corners of an object are to be located. Image should be a valid image. It should not be empty. boundingBox: The bounding box coordinates where the corner finding is to be confined. Content of this variable, i.e., boundingBox. pos and boundingBox .size should be valid values. boundingBox .size.rows and boundingBox .size.cols should not be 0. cornersInObject: The located corner pair if finding is successful. This is an output parameter. This should be a non-null object.
Return:	API execution status.
Description:	A pair of corners is searched for within the object in the specified bounding box region.

Example:	FOIL::CImage<CGrayImage8, GRAY8> objectImage;
	FOIL::CRectangle<int,int> minBoundingBox;
	FOIL::CCornerPair<unsigned long> corners;
	FOIL::COrient<int,int, CGrayImage8, GRAY8,int> orientObj;
	FOIL::err_ret return;
	
	return = orientObj . FindCorner (objectImage, minBoundingBox, corners);

4.4 Required Classes for the Measurement and Fit APIs

Class:	**CMeasure Class**
Description:	This is a template-based class that will expose the APIs to the user to

Class:	**CMeasure Class**
	make measurement by extracting pointsets and fitting the shapes into the points.

Syntax:	template < class imageClass, int imgTyp, class pointTyp, class dataType>
	CMeasure
	{
	public:
	static CPointSet<pointTyp> Measure (CMeasurementTool<pointTyp> measurementTool, CImage<imageClass, imgTyp> image);
	
	static err_ret Fit (CPointSet<pointTyp> pointSet, CLineSegment<pointTyp>& lineSegment);
	
	static err_ret Fit (CPointSet<pointTyp> pointSet, CCircle<pointTyp>& circle);
	static CPoint<pointTyp> CMeasure::DetectEdge(CLineSegment<pointTyp> lineSegment, CImage<imageClass, imgTyp> image);
	}

Template Parameters	ImageClass: Input should be of Cimage class type e.g.: CMatrix8, CGrayImage8, CGrayImage16, etc. ImgType: Input should be of integer type e.g.: BLACKWHITE, GRAY8, etc. PointTyp: Input should be of type int, floatlong, etc. This is the input for the point type. DataType: The data type representing the pixel data type of the image. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**Measure**: This api will extract the point sets from the user-defined measurement tool. The measurement tool will be defined in terms of image coordinates. This api will perform a single pixel edge detection along the defined line segments and returns the point sets. **Fit**: This api will expose the functionality to fit a specified geometric object in the set of collected points. There are different overloaded APIs for fitting different shapes. **DetectEdge:** This private member function of CMeasure will extract the edge detected on the given image.

4.5 Measurement and Fit API's

1. CMeasure::Measure()
2. CMeasure::Fit()
3. Cmeasure::DetectEdge()

Class:	**CPointSet Class**
Description:	This template-based class will represent the point set which is basically a collection of points.

Syntax:	
	```
	template <class pointTyp>
	class CPointSet
	{
	        private:
	                int pointCount;
	                std::vector< CPoint <pointTyp>> PointList;
	        public:
	                void AddPointToSet (CPoint<pointTyp> point);
	                {
	                        PointList.push_back(point);
	                        pointCount++;
	                }
	                int GetNumPoints(void)
	                {
	                        return (pointCount);
	                }
	                 CPoint<pointType> GetPoint(int Index)
	                {
	                    if (Index >= segmentCount) && (Index
	        <=segmentCount)
	                        return (PointList[Index]);
	                }
	}
	```

Template	pointTyp: Input should be the data type of the coordinate locations. It is

Class:	**CPointSet Class**
Parameters	dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**AddPointToSet**: adding new points to the existing set. **GetNumPoints**: extracts the point count stored in the class. **GetPoint**: extracts the stored point from a specified index in the list.

Summary:	**Extracts the measurement points**
Syntax:	CPointSet<pointTyp> CMeasure::Measure (CMeasurementTool<pointTyp> measurementTool, CImage<imageClass, imgTyp> image);
Arguments:	measurementTool: The measurement tool specified by user that is to be used for extracting point sets from the image. This should be a non-null object. image: The input image upon which the measurement tools are placed. Image should be a valid image. It should not be empty.
Return:	Point set → contains collection of points detected by measurement tool.
Description:	Single pixel edge detection is attempted on each of the line segments defined in the toolset. The resultant point set can be used to fit the respective geometric object and extract its measurements.

Example:	FOIL::CMeasurementTool<long> toolset;
	FOIL::CPointSet<long> points;
	FOIL::CImage<CGrayImage8, GRAY8> objectImage;
	FOIL::CMeasure< CGrayImage8, GRAY8,long,long> measuringObj;
	
	points = measuringObj . Measure (toolset, objectImage);

Summary:	**Fits a line on the point set**
Syntax:	err_ret CMeasure::Fit (CPointSet<pointTyp> pointSet, CLineSegment<pointTyp>& lineSegment);
Arguments:	pointSet: The point set extracted by the measure process using the user-defined measurement tool.Content of this point set should not be negative. lineSegment: The best fit line in the given point set. This is an output parameter. This should be a non-null object.
Return:	Best-fit line segment from the given set of points.
Description:	A line fitting algorithm is attempted on the collection of points provided. The line segment that fits the maximum number of points from the point set is returned.

Example:	FOIL::CPointSet<long> points;
	FOIL::CLineSegment<long> FittedLine;

Summary:	**Fits a line on the point set**

	FOIL::CMeasure< CGrayImage8, GRAY8,long,long> measuringObj;
	FOIL::err_ret return;
	
	return = measuringObj . Fit (points, FittedLine);

Summary:	**Fits a circle on the point set**
Syntax:	err_ret CMeasure::Fit(CPointSet<pointTyp> pointSet, CCircle<pointTyp>& circle);
Arguments:	pointSet: The point set extracted by the measure process using the user-defined measurement tool. Content of this point set should not be negative. Circle: The best fit circle in the given point set. This is an output parameter. This object should be non-null.
Return:	Best-fit circle from the given set of points.
Description:	A circle fitting algorithm is attempted on the collection of points. The circle that fits the maximum number of points from the point set is returned.

Example:	FOIL::CPointSet<long> points;
	FOIL::CCircle<long> FittedCircle;
	FOIL::CMeasure< CGrayImage8, GRAY8,long,long> measuringObj;
	FOIL::err_ret return;
	
	return = measuringObj . Fit (points, FittedCircle);

Summary:	**Detects the Edge in the input image**
Syntax:	static CPoint<pointTyp> CMeasure::DetectEdge(CLineSegment<pointTyp> lineSegment, CImage<imageClass, imgTyp> image);
Arguments:	lineSegment: The line segment along which the single pixel edge detection is to be done. Content of this lineSegment should not be negative and the value for startpoint should not be equal to endpoint. Image: The image upon which the edges are to be located. Image should be a valid image. It should not be empty.
Return:	This api will return the pixel coordinates along the line segment where the edge was detected in the image.
Description:	Detects the edge touching the line segment, which is passed as the parameter from the input image.

Example:	FOIL::CPointSet<long> point;
	FOIL::CLineSegment<long> lineSegment
	FOIL::CImage< CGrayImage8, GRAY8> image;
	FOIL::CMeasure< CGrayImage8, GRAY8,long,long> measuringObj;
	point = measuringObj .DetectEdge(lineSegment, image);

4.6 Required Classes for the Spatial Metric Tool APIs

Class:	**CSpatialMetricTools Class**
Description:	This is a template-based class that will extract the spatial metrics from the measured image coordinates.

Syntax:	template<class pointTyp, int rowHoles, class imageClass, int imgTyp >
	 CSpatialMetricTools
	 {
	 public:
	 static float XLenght (CLineSegment<pointTyp> lineSegment, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
	 static float YLength(CLineSegment<pointTyp> lineSegment, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
	 static float Length(CLineSegment<pointTyp> lineSegment, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
	 static float XAngle(CLineSegment<pointTyp> lineSegment);
	 static float YAngle(CLineSegment<pointTyp> lineSegment);
	 static float Angle(CLineSegment<pointTyp> lineSegment1, CLineSegment<pointTyp> lineSegment2);
	 static float Angle(CArc<pointTyp> arc);
	 static float Diameter(CCircle<pointTyp> circle, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
	 static float Radius(CCircle<pointTyp> circle, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
	 static float Area(CRegion<pointTyp> region, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
	 }

Template Parameters	PointTyp: Input should be of type int, floatlong, etc. This is the input for the point type.

Class:	**CSpatialMetricTools Class**
	rowHoles: Input to this template variable should be a integer value, i.e., count of rows. ImageClass:Input should be of Cimage class type e.g.:-CGrayImage8, CGrayImage16, etc. ImgType: Input should be of integer type e.g.: GRAY8m, GRAY16, etc.
Member Functions:	**XLength:** This api will get the x-coordinate length between the line segment endpoints in real world measurement units. **YLength:** This api will get the y-coordinate length between the line segments endpoints in real world measurement units. **Length:** This api will return the absolute length of the line segment in terms of real world measurement units. **XAngle:** This api will give the angle in degrees made by the line segment. w.r.t the X axis. **YAngle:** – This api will give the angle in degrees made by the linesegment, w.r.t the Y-axis. **Angle:** This api will return the angle in degrees made by the linesegment2 sweeping in clock wise direction from linesegment1. **Diameter:** This api will return the diameter of the given circle in real world units. **Radius:** This api will return the radius of the given circle in real world units. **Area:** This api will return the computed area of the given region in real world units.

4.7 Spatial Metric Tool API's

1. CSpatialMetricTools::XLength()
2. CSpatialMetricTools::YLength()
3. CSpatialMetricTools::Length()
4. CSpatialMetricTools::XAngle()
5. CSpatialMetricTools::YAngle()
6. CSpatialMetricTools::Angle()
7. CSpatialMetricTools::Diameter()
8. CSpatialMetricTools::Radius()
9. CSpatialMetricTools::Area()

Summary:	**Finds the length of a line segment along X-axis**

Summary:	**Finds the length of a line segment along X-axis**
Syntax:	float CSpatialMetricTools::XLength (CLineSegment<pointTyp> lineSegment, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
Arguments:	lineSegment: The line segment whose length w.r.t X-axis is to be determined. Content of this linesegment should not be negative and value for startpoint should not be equal to endpoint. ordSystem – The coordinate system defined by the user across which he want to make the measurement. This should be a non-null object.
Return:	Length of the line segment along X-axis in physical units.
Description:	The distance between the end points of the line segment along X-axis is computed. This is converted to the real world measurement units using the specified coordinate system.

Example:	FOIL::CLineSegment<int> line;
	FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float length;
	
	length = SpatialTool . Xlength (line, ordinate10by10);

Summary:	**Finds the length of a line segment along Y-axis**
Syntax:	float CSpatialMetricTools:: YLength(CLineSegment<pointTyp> lineSegment, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
Arguments:	lineSegment: The line segment whose length w.r.t Y-axis is to be obtained. Content of this lineSegment should not be negative. And value for startpoint should not be equal to endpoint. ordSystem: The coordinate system defined by the user across which he wants to make the measurement. This should be a non-null object.
Return:	Length of the line segment along Y-axis in physical units.
Description:	The distance between the end points of the line segment along Y-axis is computed. This is converted to the real world measurement units using the specified coordinate system.

Example:	FOIL::CLineSegment<int> line;
	FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float length;
	
	length = SpatialTool . Ylength (line, ordinate10by10);

Summary:	**Finds the absolute length of a line segment**
Syntax:	float CSpatialMetricTools:: Length(CLineSegment<pointTyp> lineSegment, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
Arguments:	lineSegment: The line segment whose absolute length. Content of this lineSegment should not be negative And value for startpoint should not be equal to endpoint. ordSystem: The coordinate system defined by the user across which he wants to make the measurement. This should be a non-null object.
Return:	Absolute length of the line segment.
Description:	The absolute distance between the end points of the line segment is computed. This is converted to the real world measurement units using the specified coordinate system.

Example:	FOIL::CLineSegment<int> line;
	FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float length;
	
	length = SpatialTool . Length (line, ordinate10by10);

Summary:	**Finds the angle made by a line segment with X-axis.**
Syntax:	float CSpatialMetricTools::XAngle(CLineSegment<pointTyp> lineSegment);
Arguments:	lineSegment: The line segment whose slope is to be measured across the X-axis. Content of this lineSegment should not be negative. And value for startpoint should not be equal to endpoint.
Return:	Line-segment's X-axis slope in degrees.
Description:	The slope of the line segment to the X-axis is computed using the two point line representation.

Example:	FOIL::CLineSegment<int> line;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float angle;
	
	angle = SpatialTool . XAngle (line);

Summary:	**Finds the angle made by a line segment with Y-axis.**
Syntax:	float CSpatialMetricTools::YAngle(CLineSegment<pointTyp> lineSegment);
Arguments:	lineSegment: The line segment whose slope is to be measured across the Y-axis. Content of this lineSegment should not be negative and value for

	startpoint should not be equal to endpoint.
Return:	Line-segment's Y-axis slope in degrees.
Description:	The slope of the line segment to the Y-axis is computed using the two point line representation.

Example:	FOIL::CLineSegment<int> line;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float angle;
	
	angle = SpatialTool . YAngle (line);

Summary:	**Finds the angle between a pair of line segments.**
Syntax:	float CSpatialMetricTools::Angle(CLineSegment<pointTyp> lineSegment1, CLineSegment<pointTyp> lineSegment2);
Arguments:	lineSegment1: The first line segment among the pair whose angle is to be computed. Content of this lineSegment should not be negative and value for startpoint should not be equal to endpoint. lineSegment2: The second line segment among the pair whose angle is to be computed. Content of this lineSegment should not be negative and value for startpoint should not be equal to endpoint.
Return:	Angle in degrees between the two line segments.
Description:	Angle between the two line segments is computed through a sweep in the clockwise direction.

Example:	FOIL::CLineSegment<int> line1;
	FOIL::CLineSegment<int> line2;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float angle;
	
	angle = SpatialTool . XAngle (line1, line2);

Class:	**CArc Class**
Description:	This is a template-based class that will represent an arc in terms of its center and radius and arc end points

Syntax:	template <class pointTyp>
	CArc:: public CCircle<pointTyp>
	{
	    public:
	        CPoint<pointTyp> arcStart;
	        CPoint<pointTyp> arcEnd;
	}

Class:	**CArc Class**
Template Parameters	pointTyp: Input data type should match that of the coordinate locations. It is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Summary :	**Finds the inclusive angle made by an arc.**
Syntax:	float CSpatialMetricTools::Angle(CArc<pointTyp> arc);
Arguments :	Arc: The arc whose inclusive angle is to be found. It should contain non-negative coordinates within the image boundaries.
Return:	Angle in degrees made by the arc.
Description :	An arc is represented as a pair of lines with a common end point. This API finds the angle between these two lines through a sweep in the clockwise direction.

Example:	FOIL::CArc<int> arc;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float angle;
	
	angle = SpatialTool . Angle (arc);

Class:	**CCircle Class**
Description:	This is a template-based class that will represent a circle in terms of its center and radius

Syntax:	template <class pointTyp>
	CCircle
	{
	public:
	long radius;
	CPoint<pointTyp> center;
	}

Template Parameters	pointTyp: Input data type should match that of the coordinate locations. It is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Summary:	**Gets the diameter of a circle.**
Syntax:	float CSpatialMetricTools::Diameter(CCircle<pointTyp> circle, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
Arguments:	Circle: The circle whose diameter is to be found. Circle should have points defined within the image boundaries and the radius within the image boundaries. ordSystem: The coordinate system defined by the user across which he wants to make the measurement. This should be a non-null object.
Return:	Diameter of the circle in real world units.
Description:	A circle is defined by its center and radius. This API converts the radius to its diameter, and returns the value in real world units using the specified coordinate system.

Example:	FOIL::CCircle<int> circle;
	FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float diameter;
	
	diameter = SpatialTool . Diameter (circle, ordinate10by10);

Summary:	**Gets the radius of the circle.**
Syntax:	float CSpatialMetricTools::Radius(CCircle<pointTyp> circle, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
Arguments:	Circle should be have points defined within the image boundaries and the radius within the image boundaries. ordSystem: The coordinate system defined by the user across which he wants to make the measurement. This should be a non-null object
Return:	Radius of the circle in real world units.
Description:	A circle is specified by its center and radius. This API converts the radius to its real world value using the specified coordinate system.

Example:	FOIL::CCircle<int> circle;
	FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float radius;
	
	radius = SpatialTool . Radius (circle, ordinate10by10);

Class:	**CRegion Class**
Description:	This is a template-based class that will represent a region in terms of successive points.

Syntax:	
	```
	template <class pointTyp>
	CRegion::CPointSet<pointTyp>
	{
	        private:
	                int pointCount;
	                std::vector< CPoint <pointTyp>> PointList;
	        public:
	                void AddPointToSet (CPoint<pointTyp> point);
	                {
	                        PointList.push_back(point);
	                        pointCount++;
	                }
	                int GetNumPoints(void)
	                {
	                        return (pointCount);
	                }
	                 CPoint<pointType> GetPoint(int Index)
	                {
	                        if (Index >= segmentCount) && (Index
	        <=segmentCount)
	                        return (PointList[Index]);
	                }
	}
	```

Template Parameters	pointTyp: Input data type should match that of the coordinate locations. It is dependent on the image dimensions. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**AddPointToSet**: adding new points to the existing set. **GetNumPoints**: extracts the point count stored in the class. **GetPoint**: extracts the stored point from a specified index in the list.

Summary:	**Gets the area of a region.**
Syntax:	float CSpatialMetricTools::Area(CRegion<pointTyp> region, CCoordinateSsystem< rowHoles, pointTyp, imageClass , imgTyp > ordSystem);
Arguments:	region: The region whose area is to be found. The point coordinates should be within the image boundaries. ordSystem: The coordinate system defined by the user across which he wants to make the measurement. This should be a non-null object.
Return:	Area of the region in real world units.
Description:	A region is assumed to be a set of points that form a polygon. This API computes the area of the resulting polygon and returns the value in real world units.

Example:	FOIL::CRegion<int> searchArea;
	FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10;
	FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool;
	float area;
	
	area = SpatialTool . Area (searchArea, ordinate10by10);

4.8 Required Classes for the Photometric Tool APIs

Class:	**CPhotoMetricTools Class**
Description:	This is a template-based class that will extract the photo metrics from the input image.

Syntax:	template<class colourTyp, class pointTyp, class imageClass, int imgTyp>
	CPhotoMetricTools
	{
	static CColour<colourTyp> GetAverageColour(CRegion<pointTyp> region, CColourImage<imageClass, imgTyp> image);
	
	static float GetAverageGray(CRegion<pointTyp> region, CImage<imageClass, imgTyp> image);
	}

Template	ColourTyp:-Input can be of unsigned float, unsigned int, unsigned long,

Parameters	unsigned char, etc. PointTyp: Input should be of type int,floatlong, etc. ,this is the input for the point type. ImageClass:Input should be of Cimage class type e.g.: CMatrix8, CGrayImage8, CGrayImage16, etc. ImgType: Input should be of integer type e.g.: BLACKWHITE, GRAY8, etc.
Member Functions:	**GetAverageColour:** – This api will extract the average value of each colour channel in the specified region **GetAverageGray:** – This api will extract the average value of gray scale values in the specified region

4.9 Photometric Tool API's

1. CPhotoMetricTools::GetAverageColour()
2. CPhotoMetricTools::GetAverageGray()

Class:	**CColour Class**
Description:	This is a template-based class that will represent color in terms of its Red, Green and Blue intensities.

Syntax:	template <class colourTyp>
	CColour
	{
	public:
	colourTyp Red;
	colourTyp Green;
	colourTyp Blue;
	}

Template Parameters	colourTyp: Input should be of the data type of pixel width. e.g.: unsigned char, unsigned long, unsigned int, etc.
Member Functions:	**None**

Summary:	**Gets the average color of a region in the image.**
Syntax:	CColour<colourTyp> CPhotoMetricTools::GetAverageColour(CRegion<pointTyp> region,

	CColourImage<imageClass, imgTyp> image);
Arguments:	region: The region of the image whose average color value is to be computed. The size of the region should not be zero and it should contain coordinates within the image boundaries. image: This is the input colour image upon the region is specified. The input image should be a valid image and should not be empty.
Return:	Color object.
Description:	A region is assumed to be a set of points that form a polygon. This API computes the average intensity of each color channel and returns a color object with the average intensities.

Example:	FOIL::CRegion<int> searchArea;
	FOIL::CColour<int> AverageColour;
	FOIL::CColourImage< CRGBPlanar8, RGBPLANAR8> inputImage;
	FOIL::CPhotoMetricTools<int, int, CRGBPlanar8, RGBPLANAR8> PhotometricTool;
	
	AverageColour = PhotometricTool . GetAverageColour (searchArea, inputImage);

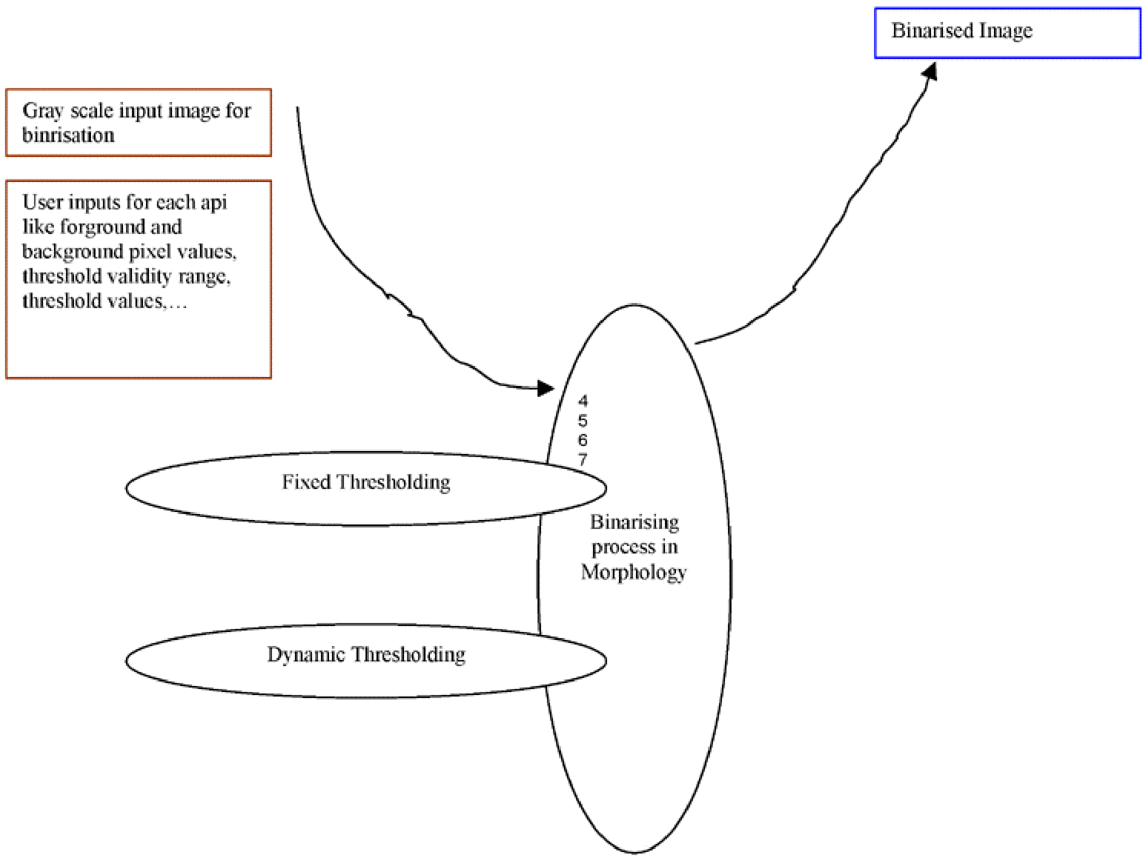
Summary:	**Gets the average gray scale value of a region in the image.**
Syntax:	float CPhotoMetricTools::GetAverageGray (CRegion<pointTyp> region, CImage<imageClass, imgTyp> image);
Arguments:	region: The region of the image whose average gray value is to be computed. The size of the region should not be zero and it should contain coordinates defined within the image boundaries. image: This is the input gray image upon which the region is specified. The input image should be a valid image and should not be empty.
Return:	Average gray scale value of pixels in the image specified within the region.
Description:	A region is assumed to be a set of points that form a polygon. This API computes the average of gray scale values for all pixels within the region of the image.

Example:	FOIL::CRegion<int> searchArea;
	FOIL::CImage< CGrayImage8, GRAY8> inputImage;
	FOIL::CPhotoMetricTools<int, int, CGrayImage8, GRAY8> PhotometricTool;
	float averageGray;
	
	averageGray = PhotometricTool . GetAverageGray (searchArea, inputImage);

5 MVIL LIBRARY: MORPHOLOGY

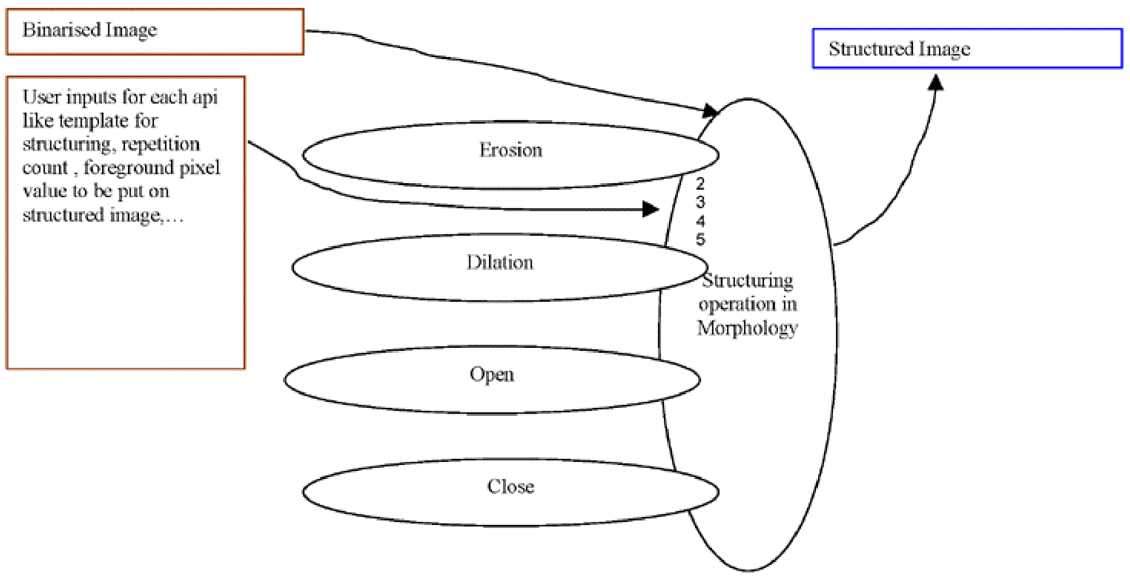
5.1 Control/DataFlows Diagrams

Binarising

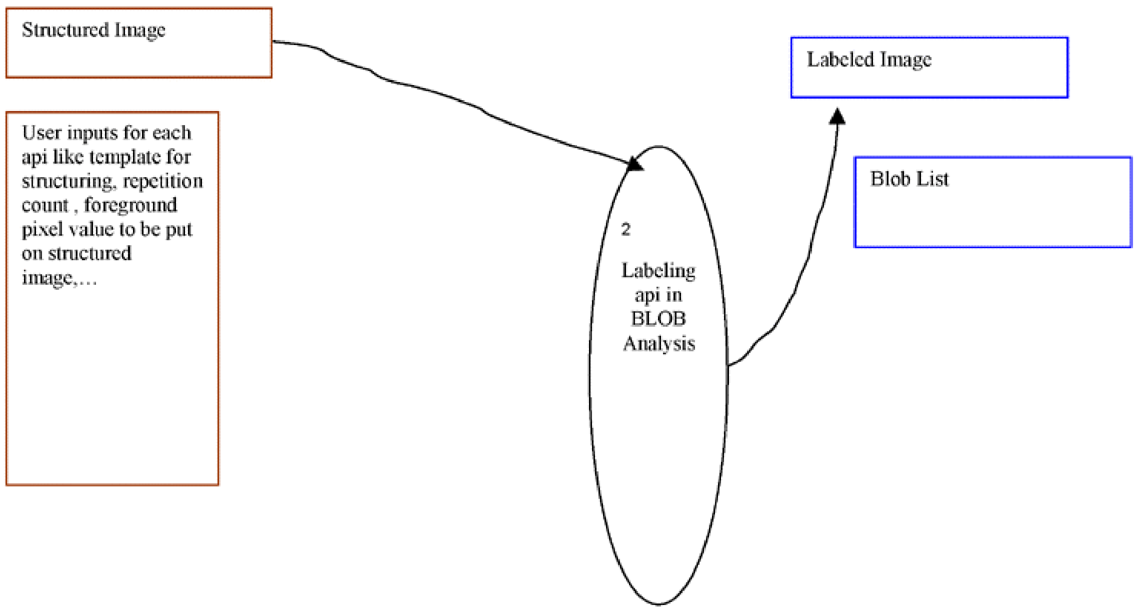

Binarised Image
Gray scale input image for binrisation
User inputs for each api like forground and background pixel values, threshold validity range, threshold values,…
4
5
6
7
Fixed Thresholding
Binarising process in Morphology
Dynamic Thresholding

●	- Library apis
🟫	- user inputs
🟦	- intermediate / outputs

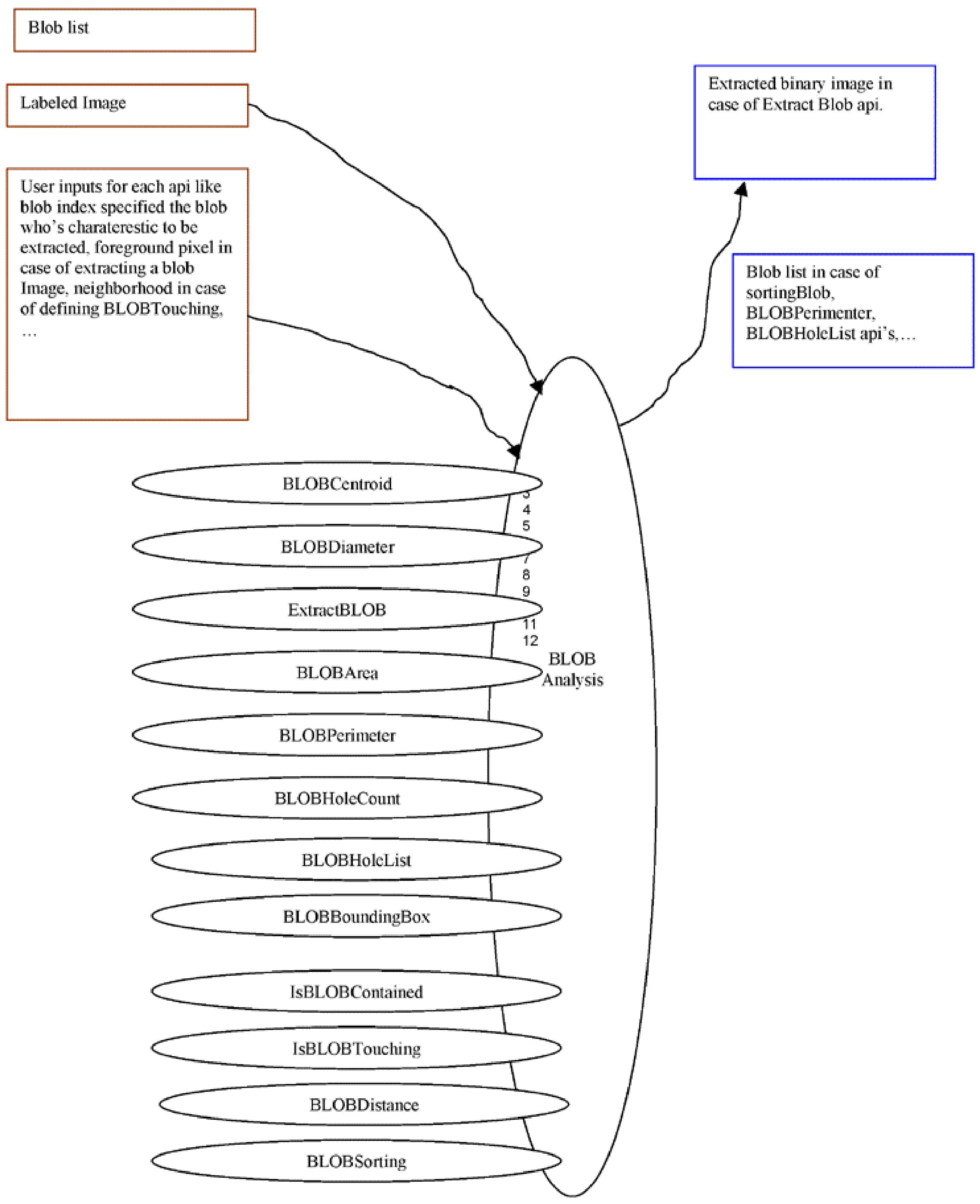
Structuring


Binarised Image
User inputs for each api like template for structuring, repetition count , foreground pixel value to be put on structured image,…
Erosion
Dilation
Open
Close
2
3
4
5
Structuring operation in Morphology
Structured Image

Labeling


Structured Image
User inputs for each api like template for structuring, repetition count , foreground pixel value to be put on structured image,…
2
Labeling api in BLOB Analysis
Labeled Image
Blob List

BLOB Analysis


Blob list
Labeled Image
User inputs for each api like blob index specified the blob who's charaterestic to be extracted, foreground pixel in case of extracting a blob Image, neighborhood in case of defining BLOBTouching, …
Extracted binary image in case of Extract Blob api.
Blob list in case of sortingBlob, BLOBPerimenter, BLOBHoleList api's,…
BLOBCentroid
BLOBDiameter
ExtractBLOB
BLOBArea
BLOBPerimeter
BLOBHoleCount
BLOBHoleList
BLOBBoundingBox
IsBLOBContained
IsBLOBTouching
BLOBDistance
BLOBSorting
BLOB Analysis

This section details the API's built for Morphology within the MViL library.

5.2 Required Classes for the Thresholding APIs

Class:	**CBinarize Class**

Class:	**CBinarize Class**
Description:	This template class will expose various thresholding APIs used to binarize a gray sale image.

Syntax:	template <class inImageClass, int inImgTyp, class threshImageClass, int threshImgTyp, class outImageClass, int outImgTyp>
	class CBinarize
	{
	public:
	static err_ret FixedThreshold (CImage<inImageClass , inImgTyp> inputImage, int lowThreshold, int highThreshold, range_type rangeSelection, int foregroundPixel , int backgroungPixel, CImage<outImageClass , outImgTyp> * outputImage);
	static err_ret DynamicThreshold (CImage<inImageClass , inImgTyp> inputImage, CImage<threshImageClass, threshImgTyp> lowThreshold, CImage< threshImageClass, threshImgTyp > highThreshold, range_type rangeSelection, int foregroundPixel, int backgroundPixel, CImage<outImageClass , outImgTyp> * outputImage);
	}

Template Parameters	inImageClass: the class type of the input image Ex: CGrayScale8, CGrayScale16, … inImgTyp: the image type of the input image Ex: GRAY8, GRAY16, … threshImageClass: the class type of the threshold image Ex: CGrayScale8, CGrayScale16, … threshImgTyp: the image type of the threshold image Ex: GRAY8, GRAY16, … OutImageClass: the class type of the output image Ex: CGrayScale8, CGrayScale16, … OutImgTyp: the image type of the output image Ex: GRAY8, GRAY16, …
Member Functions:	**FixedThreshold:** This template function will binarize the input image based on a pair of constant thresholds across all pixels. **DynamicThreshold:-**This template function will binarize the input image based on a pair of thresholds defined for each individual pixel.

5.3 Thresholding API's

1. CBinarize::FixedThreshold()
2. CBinarize::DynamicThreshold()

Summary:	**Applies fixed thresholding on the gray scale input image to binarize it.**
Syntax:	err_ret CBinarize::FixedThreshold (CImage<inImageClass , inImgTyp> inputImage, int lowThreshold, int highThreshold, range_type rangeSelection, int foregroundPixel , int backgroungPixel, CImage<outImageClass , outImgTyp> * outputImage);
Arguments:	inputImage: This is the gray scale image that is to be binarized. The input image should be a valid gray scale image and should not be empty. lowThreshold: This is the lower bound of the threshold. Should be a positive number. highThreshold: This is the upper bound of the threshold. Should be a positive number. rangeSelection: This specifies whether to put 1 in output pixel if the input pixel value falls within the specified threshold range or outside it. It should take values from FOIL enumerated types RANGE_WITHIN_THRESHOLD and RANGE_OUTSIDE_THRESHOLD. foregroundPixel: This is the pixel value considered if the input image pixel value is within the threshold bounds when the rangeSelection is WITHIN_BOUNDS else backgroundPixel. This valid range is from 0 to max of pixel type. backgroundPixel: This is the pixel value considered if the input image pixel value is outside the threshold bounds when the rangeSelection is OUTSIDE_BOUNDS else foregroundPixel. This valid range is from 0 to max of pixel type. outputImage: This is the resultant image of binarization. This should be a non negative image
Return:	API execution status.

Description:	`if(rangeSelection = =  RANGE_WITHIN_THRESHOLD)`
	`{`
	`    within = foregroundPixel;`
	`     without = backgroundPixel;`
	`}`
	`   else`
	`{`
	`    within = backgroundPixel;`
	`     without = foregroundPixel;`
	`}`
	For each pixel value.
	if ((inputImage[i][j] >= lowThreshold) && (inputImage[i][j] <=

Summary:	**Applies fixed thresholding on the gray scale input image to binarize it.**

	highThreshold))
	    outputImage[i][j] = within;
	  else
	    outputImage[i][j] = without;

Example:	FOIL::CImage<CGrayImage8, GRAY8> inputGrayImage;
	FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage;
	  int lowThreshold = 40;
	 int highThreshold = 70;
	int foregroundPixel = 100;
	int backgroundPixel = 0;
	FOIL::CBinarize< CGrayImage16, GRAY16, CGrayImage8, GRAY8, CGrayImage8, GRAY8> Thresholding;
	FOIL::err_ret status=-1;
	
	status = Thresholding . FixedThreshold(inputGrayImage, lowThreshold , highThreshold, FOIL:: RANGE_OUTSIDE_THRESHOLD, foregroundPixel, backgroundPixel, BinarizedImage);

Summary:	**Applies dynamic thresholding on the gray scale input image to binarize it.**
Syntax:	err_ret CBinarize::DynamicThreshold (CImage<inImageClass , inImgTyp> inputImage, CImage< threshImageClass, threshImageTyp > lowThreshold, CImage< threshImageClass, threshImageTyp> highThreshold, range_type rangeSelection, int foregroundPixel, int backgroundPixel, CImage<outImageClass , outImgTyp> * outputImage);
Arguments:	inputImage: This is the gray scale image that is to be binarized. The input image should be a valid grayscale image and should not be empty. lowThreshold: This is the array of lower bound of the threshold for each pixel in the input image. Should be a positive number. highThreshold: This is the array of upper bound of the threshold for each pixel in the input image. Should be a positive number. rangeSelection: This specifies whether to set output pixel value to 1 if the input pixel value falls within the specified threshold range or outside it. It should take values from FOIL enumerated types RANGE_WITHIN_THRESHOLD and RANGE_OUTSIDE_THRESHOLD. foregroundPixel: This is the pixel value to be retained if the input image pixel value is within the threshold bounds when the rangeSelection is WITHIN_BOUNDS else backgroundPixel. This valid range is from 0 to max of pixel type backgroundPixel: This is the pixel value to be retained if the input image

Summary:	**Applies dynamic thresholding on the gray scale input image to binarize it.**
	pixel value is outside the threshold bounds when the rangeSelection is OUTSIDE_BOUNDS else foregroundPixel. This valid range is from 0 to max of pixel type outputImage: This is the resultant image of binarization. This should not be empty.
Return:	API execution status.

Description:	if(rangeSelection = = RANGE_WITHIN_THRESHOLD)
	{
	within = foregroundPixel;
	without = backgroundPixel;
	}
	else
	{
	within = backgroundPixel;
	without = foregroundPixel;
	}
	For each pixel value.
	
	if ((inputImage[i][j] >= lowThreshold[i][j]) && (inputImage[i][j] <= highThreshold[i][j]))
	outputImage[i][j] = within;
	else
	outputImage[i][j] = without;

Example:	FOIL::CImage<CGrayImage16, GRAY16> inputGrayImage;
	FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage;
	FOIL::CImage< CGrayImage16, GRAY16> lowThreshold;
	FOIL::CImage< CGrayImage16, GRAY16> highThreshold;
	int foregroundPixel = 100;
	int backgroundPixel = 0;
	FOIL::CBinarize< CGrayImage16, GRAY16, CGrayImage16, GRAY16, CGrayImage8, GRAY8> Thresholding;
	FOIL::err_ret status=-1;
	
	status = Thresholding . DynamicThreshold(inputGrayImage, lowThreshold , highThreshold, FOIL:: RANGE_WITHIN_THRESHOLD, foregroundPixle, backgroundPixle, BinarizedImage);

5.4 Required Classes for the Structuring APIs

Class:	**CStructureElement Class**

Class:	**CStructureElement Class**
Description:	This template class will expose various structuring APIs used in morphology operations.

Syntax:	template <class ImageClass, int ImgTyp>
	class CStructureElement
	{
	public:
	static err_ret Erosion (CImage<ImageClass , ImgTyp> inputImage, template_type templateSelection, int foregroundPixel, int backgroundPixel, CImage<ImageClass , ImgTyp> * outputImage);
	
	static err_ret Dilation (CImage<ImageClass , ImgTyp> inputImage, template_type templateSelection, int foregroundPixel, int backgroundPixel, CImage<ImageClass , ImgTyp> * outputImage);
	
	static err_ret Open (CImage<ImageClass , ImgTyp> inputImage, template_type templateSelection, int repetitionFactor, int foregroundPixel, int backgroundPixel, CImage<ImageClass , ImgTyp> * outputImage);
	
	static err_ret Close (CImage<ImageClass , ImgTyp> inputImage, template_type templateSelection, int repetitionFactor, int foregroundPixel, int backgroundPixel, CImage<ImageClass , ImgTyp> * outputImage);
	}

Template Parameters	ImageClass: Input should be of Cimage class type e.g.: CMatrix8, CGrayImage8, CGrayImage16, etc. ImgType: Input should be of integer type e.g.: BLACKWHITE, GRAY8, etc.
Member Functions:	**Erosion:** This template function will erode the input image as per the selected standard template. **Dilation:** This template function will dilate the input image as per the selected standard template. **Open**: This template function will open the input image as per the

Class:	**CStructureElement Class**
	selected standard template. **Close**: This template function will close the input image as per the selected standard template.

5.5 Structuring API’s

1. CStructureElement::Erosion()
2. CStructureElement::Dilation()
3. CStructureElement::Open()
4. CStructureElement::Close()

Summary:	**Applies Erosion on the binarized image with a specified template.**
Syntax:	err_ret CStructureElement::Erosion (CImage<ImageClass , ImgTyp> inputImage, template_type templateSelection, int foregroundPixel, int backgroundPixel, CImage<ImageClass , ImgTyp> * outputImage);
Arguments:	inputImage: This is the binarized image that is to be eroded with the specified template.It should be a valid input image and should not be empty. templateSelection: This is the type of NbyN template that will be used to erode the input image. It assumes the values from FOIL enumerated types TEMPLATE3x3, TEMPLATE5x5 and TEMPLATE7x7. foregroundPixel: This is the pixel value which is to be compared across the template and put if the all the pixels in the template comparison satisfies. This should be the same foreground pixel value used for binarization. backgroundPixel: This is the pixel value which is to be compared across the template and put if none of the pixels in the template comparison succeeds. This should be the same foreground pixel value used for binarization. outputImage: This is the resultant image of erosion. This should be non-null.
Return:	API execution status.

Description:	The input image should have been binarized.
	API performs the following operations to each pixel value.
	if (inputImage[i][j] = = foregroundPixel for all corresponding pixels having 1 in the selected template)
	    outputImage[i][j] = foregroundPixel;
	else
	    outputImage[i][j] = backgroundPixel;

Example:	FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage;

Summary:	**Applies Erosion on the binarized image with a specified template.**

	FOIL::CImage<CGrayImage8, GRAY8> ErodedImage;
	int foregroundPixel = 100;
	int backgroundPixel = 0;
	FOIL::CStructureElement< CGrayImage8, GRAY8> Structuring;
	FOIL::err_ret status=-1;
	
	status = Structuring . Erosion(BinarizedImage, FOIL:: TEMPLATE3x3, foregroundPixel, backgroundPixle, ErodedImage);

Summary:	**Applies Dilation on the binarized image with a specified template.**
Syntax:	err_ret CStructureElement::Dilation (CImage<ImageClass , ImgTyp> inputImage, template_type templateSelection, int foregroundPixel, int backgroundPixel, age<ImageClass , ImgTyp> * outputImage);
Arguments:	inputImage: This is the binarized image that is to be dilated with the specified template.This should be a valid input image and should not be empty. template: This is the type of NbyN template that will be used to dilate the input image. It assumes the values from FOIL enumerated types TEMPLATE3x3, TEMPLATE5x5 and TEMPLATE7x7. foregroundPixel: Pixel value that is compared across all template pixels and retained if there is at least one match. This should be the same foreground pixel value used for binarization. backgroundPixel: This is the pixel value which is to be compared across the template and put if none of the pixels in the template comparison succeeds. This should be the same foreground pixel value used for binarization. outputImage: This is the resultant image of dilation. This should be non-null.
Return:	API execution status.

Description:	The input image should have been binarized.
	API performs the following operations to each pixel value.
	if (inputImage[i][j] = = foregroundPixel for any one of corresponding pixels having 1 in the selected template)
	    outputImage[i][j] = foregroundPixel;
	else
	    outputImage[i][j] = backgroundPixel;

Example:	FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage;
	FOIL::CImage<CGrayImage8, GRAY8> DilatedImage;
	int foregroundPixel = 100;
	int backgroundPixel = 0;
	FOIL::CStructureElement< CGrayImage8, GRAY8> Structuring;

Summary:	**Applies Dilation on the binarized image with a specified template.**

	FOIL::err_ret status=-1;
	
	status = Structuring . Dilation(BinarizedImage, FOIL:: TEMPLATE5x5, foregroundPixel, backgroundPixle, DilatedImage);

Summary:	**Applies Open on the binarized image with a specified template.**
Syntax:	err_ret Open (CImage<ImageClass , ImgTyp> inputImage, template_type templateSelection, int repetitionFactor, int foregroundPixle, int backgroundPixel, CImage<ImageClass , ImgTyp> * outputImage);
Arguments:	inputImage: This is the binarized image that is to be opened with the specified template. This should be a non-null object. templateSelection: This is the type of NbyN template that will be used to open the input image. It assumes the values from FOIL enumerated types TEMPLATE3x3, TEMPLATE5x5 and TEMPLATE7x7. repetitionFactor: This denotes the number of successive openings to be performed. It should be a non negative value. foregroundPixel: This is the foreground pixel value to be used in the erosion and dilation APIs. This should be the same foreground pixel value used for binarization. backgroundPixel: This is the background pixel value to be used in the erosion and dilation APIs. This should be the same foreground pixel value used for binarization. outputImage – This is the resultant image of opening. This should be non-null.
Return:	API execution status.
Description:	The input image should have been binarized. API performs the following operations: For the number of times equal to repetitionFactor do the following Call Erosion(); Call Dilation();

Example:	FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage;
	FOIL::CImage<CGrayImage8, GRAY8> * OpenedImage;
	int repetitionFactor = 5;
	int foregroundPixel = 100;
	int backgroundPixel = 0;
	FOIL::CStructureElement< CGrayImage8, GRAY8> Structuring;
	FOIL::err_ret = -1;
	
	status = Structuring . Open(BinarizedImage, FOIL:: TEMPLATE7x7, repetitionFactor, foregroundPixel, backgroundPixle, OpenedImage);

Summary:	**Applies Close on the binarized image with a specified template.**
Syntax:	err_ret Close (CImage<ImageClass , ImgTyp> inputImage, template_type templateSelection, int repetitionFactor, int foregroundPixle, int backgroundPixel, CImage<ImageClass , ImgTyp> * outputImage);
Arguments:	inputImage – This is the binarized image that is to be closed with the specified template. This should be non-null. templateSelection – This is the type of NbyN template that will be used to open the input image. It assumes the values from FOIL enumerated types TEMPLATE3x3, TEMPLATE5x5 and TEMPLATE7x7 repetitionFactor – This denotes the number of successive closing to be performed. It should be a non negative value. foregroundPixel – This is the foreground pixel value to be used in the erosion and dilation APIs. This should be the same foreground pixel value used for binarization. backgroundPixel – This is the background pixel value to be used in the erosion and dilation APIs. This should be the same foreground pixel value used for binarization. outputImage: This is the resultant image of closing. This should be non-null.
Return:	API execution status.
Description:	The input image should have been binarized. API performs the following operations: For the number of times equal to repetitionFactor do the following Call Dilation(); Call Erosion();

Example:	FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage;
	FOIL::CImage<CGrayImage8, GRAY8> * ClosedImage;
	int repetitionFactor = 5;
	int foregroundPixel = 100;
	int backgroundPixel = 0;
	FOIL::CStructureElement< CGrayImage8, GRAY8> Structuring;
	FOIL::err_ret = -1;
	
	status = Structuring . Close(BinarizedImage, FOIL:: TEMPLATE3x3, repetitionFactor, foregroundPixel, backgroundPixle, ClosedImage);

6 MVIL LIBRARY: BLOB ANALYSIS

This section details the API's built for BLOB analysis within the MViL library.

6.1 Labeling API's

1. FOIL::LabelBLOBs()

Summary:	**Performs labeling operation on an input gray image.**
Syntax:	err_ret LabelBLOBs (CImage<ImageClass , ImgTyp> inputImage, neighborhood_type templateSelection, int foregroundPixel, CImage<blobImageClass , blobImgTyp> * blobImage, CBlobList& blobList);
Arguments:	inputImage: This is the binarized image that is to be labeled to define the blobs. This should be the binarized and structured non-null object. templateSelection: This is the type of neighborhood template that will be used to define pixel touching in the input image. It assumes the values from FOIL enumerated types NEIGHBORHOOD4 and NEIGHBORHOOD8. foregroundPixel: The pixel value used while binarizing to represent the foreground of the image. blobImage: This is the output image that represents the labeled image with pixels having the label values at each respective blobs. This should be a non-null object. blobList: This is also an output parameter. It has the list of all blobs labeled and their corresponding pixel locations. This should be a non-null object.
Return:	API execution status.
Description:	API labels each pixel in the input image with a value starting with 0 if the touching defined by the template is true.

Example:	FOIL::CImage<CGrayImage8, GRAY8> inputImage;
	FOIL::CImage<CgrayImage16, GRAY16> blobImage;
	CBlobList blobList;
	int foregroundPixel = 100;
	FOIL::err_ret = -1;
	
	err_ret = FOIL::LabelBLOBs(inputImage, FOIL:: NEIGHBORHOOD4, foregroundPixel , blobImage, blobList);

6.2 Required Classes for Unary Feature Extraction APIs

Class:	**CUnaryFeature Class**
Description:	This template class will expose various APIs that will extract various unary features from the labeled output image.

Syntax:	template <class blobImageClass, int blobImgTyp>
	class CUnaryFeature
	{
	public:
	static CPoint<long> GetBLOBCentroid (CBlobList blobList, unsigned long blobIndex);
	static CBlobList GetBLOBPerimeter (CBlobList blobList, CImage<blobImageClass , blobImgTyp> blobImage, unsigned long blobIndex);
	static unsigned long GetBLOBArea (CBlobList blobList, unsigned long blobIndex);
	static err_ret ExtractBLOB (CBlobList blobList, CImage<blobImageClass, blobImgTyp> blobImage, unsigned long blobIndex, int foregroundPixle, int backgroundPixle, CImage<CGrayImage8, GRAY8> * binaryImage);
	static CBoundingBox<long> GetBLOBBoundingBox (CBlobList blobList, unsigned long blobIndex);
	static float GetBLOBDiameter (CBlobList blobList, unsigned long blobIndex);
	static long GetBLOBHoleCount (CBlobList blobList, CImage<blobImageClass , blobImgTyp> blobImage, unsigned long blobIndex);
	static CBlobList GetBLOBHoleList (CBlobList blobList, CImage<blobImageClass , blobImgTyp> blobImage, unsigned long blobIndex);
	}

Template Parameters	blobImageClass: Input should be of Cimage class type. e.g.: CGrayImage8, CGrayImage16, etc. blobImgTyp: Input should be of integer type. e.g.: GRAY8, GRAY16, etc.
Member Functions:	**GetBLOBCentriod:** This template function will find the centroid of a specified BLOB **GetBLOBPerimeter:** This template function will find the perimeter of a specified BLOB. **GetBLOBArea**: This template function will find the area of a specified BLOB **ExtractBLOB**: This template function will generate a binary image of a specified BLOB.

	GetBLOBBoundingBox: This template function will get the smallest bounding box containing the specified BLOB **GetBLOBDiameter:** This template function will get the diameter of the specified BLOB. **GetBLOBHoleCount:** – This template function will find the number of holes in the specified BLOB. **GetBLOBHoleList:** This template function will find the number of holes in the specified BLOB.

6.3 Unary Feature Extraction API's

1. CUnaryFeature::GetBLOBCentriod()
2. CUnaryFeature::GetBLOBPerimeter()
3. CUnaryFeature::GetBLOBArea()
4. CUnaryFeature::ExtractBLOB()
5. CUnaryFeature::GetBLOBBoundingBox()
6. CUnaryFeature::GetBLOBDiameter()
7. CUnaryFeature::GetBLOBHoleCount()
8. CUnaryFeature::GetBLOBHoleList()

Summary:	**Computes the centroid of a BLOB.**
Syntax:	CPoint<long> CUnaryFeature::GetBLOBCentriod (CBlobList blobList, unsigned long blobIndex);
Arguments:	blobList: This is the result of labeling operation on an input image. It has the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobIndex: It represents the index into the list. Its value should be between 1 and the size of blobList.
Return:	Pixel location which is the centroid of the specified BLOB.
Description:	API finds the centroid by averaging all pixel coordinates in a BLOB.

Example:	CBlobList blobList;
	unsigned long blobIndex = 3;
	CPoint<long> centroid;
	FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures;
	
	centroid = UnaryFeatures . GetBLOBCentroid(blobList, blobIndex);

Summary:	**Computes the perimeter of a BLOB.**
Syntax:	CBlobList CUnaryFeature::GetBLOBPerimeter (CBlobList blobList, CImage<blobImageClass, blobImgTyp> blobImage, unsigned long

Summary:	**Computes the perimeter of a BLOB.**
	blobIndex);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobIndex: It represents the index into the list. Its value should be between 1 and size of blobList.
Return:	BLOB list, which depicts the pixel co-ordinates on the perimeter of the BLOB. There can be multiple entries in the returned list if there are containments.
Description:	API gets the list of pixels that form the perimeter of the BLOB. It looks for adjacency in the blob image.

Example:	CBlobList blobList;
	CImage<CgrayImage16, GRAY16> blobImage;
	CBlobList perimeterList;
	unsigned long blobIndex = 3;
	FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures;
	
	perimeterList = UnaryFeatures . GetBLOBPerimeter(blobList, blobImage , blobIndex);

Summary:	**Computes the area of a BLOB.**
Syntax:	long CUnaryFeature::GetBLOBArea (CBlobList blobList, unsigned long blobIndex);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobIndex: It represents the index into the list. Its value should be between 1 and size of blobList.
Return:	Area of the specified blob.
Description:	API counts the number of pixels in a BLOB and returns the computed value as the area of the blob.

Example:	CBlobList blobList;
	unsigned long blobIndex = 3;
	long blobArea;
	FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures;
	
	blobArea = UnaryFeatures . GetBLOBArea(blobList, blobIndex);

Summary:	**Extracts a BLOB by generating the binary image containing that BLOB alone.**
Syntax:	err_ret CUnaryFeature::ExtractBLOB (CBlobList blobList, CImage<blobImageClass, blobImgTyp> blobImage, unsigned long blobIndex, int foregroundPixle, int backgroundPixle, CImage<CGrayImage8, GRAY8> * binaryImage);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this

Summary:	**Extracts a BLOB by generating the binary image containing that BLOB alone.**
	list should not be zero and should contain valid values. blobImage: This is the labeled image containing BLOBs. It should be a valid image and should not be empty. blobIndex: It represents the index into the list. Its value should be between 1 and size of blobList. foregroundPixel: The pixel value to be put in the foreground of the resultant binary image. backgroundPixel: The pixel value to be put in the background of the resultant binary image. binaryImage: This is the resultant binary image containing the image of blob. This should be non-null.
Return:	API execution status.
Description:	For each pixel location in the entry of blobList[blobIndex], the API will fill foregroundPixel and backgroundPixel in all other pixel locations.

Example:	CBlobList blobList;
	CImage<CGrayImage16, GRAY16> blobImage;
	CImage<CGrayImage16, GRAY16> extractedBLOB;
	char foreground Pixel = 1;
	char backgroundPixel = 0;
	unsigned long blobIndex = 3;
	FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures;
	FOIL::err_ret status = -1;
	
	status = UnaryFeatures . ExtractBLOB(blobList, blobImage, blobIndex, foregroundPixel, backgroundPixel, extractedBLOB);

Summary:	**Finds the smallest bounding Box that encloses the BLOB.**
Syntax:	CBoundingBox<long> CUnaryFeature::GetBLOBBoundingBox (CBlobList blobList, unsigned long blobIndex);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobIndex: It represents the index into the list. Its value should be between 1 and size of blobList.
Return:	CBoundingBox object that contains the BLOB.
Description:	API finds the pixel coordinates in the blob with the minimum and maximum ‘x’ and ‘y’ co-ordinates. It also finds the smallest box that can contain the blob irrespective of its orientation.

Example:	CBlobList blobList;
	CImage<long> boundingBox;
	unsigned long blobIndex = 3;
	FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures;

	boundingBox = UnaryFeatures . GetBLOBBoundingBox(blobList, blobIndex);

Summary:	**Computes the diameter of the BLOB.**
Syntax:	float CUnaryFeature::GetBLOBDiameter (CBlobList blobList, long blobIndex);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobIndex: It represents the index into the list.Its value should be between 1 and size of blobList.
Return:	Maximum distance between any two pixels in a BLOB.
Description:	API pairs different pixels in the BLOB and computes the distance between every pair. The maximum of these distances is searched for.

Example:	CBlobList blobList;
	float blobDiameter;
	long blobIndex = 3;
	FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures
	
	blobDiameter = UnaryFeatures . GetBLOBDiameter(blobList, blobIndex);

Summary:	**Computes the number of holes in a BLOB.**
Syntax:	long CUnaryFeature::GetBLOBHoleCount (CBlobList blobList, CImage<blobImageClass, blobImgTyp> blobImage, unsigned long blobIndex);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobImage: This is the labeled image containing BLOBs. It should be a valid image and should not be empty. blobIndex: It represents the index into the list. Its value should be between 1 and size of blobList.
Return:	Number of holes contained in a BLOB.
Description:	API finds the number of other labels contained in a BLOB.

Example:	CBlobList blobList;
	CImage<CGrayImage16, GRAY16> blobImage;
	long holeCount;
	unsigned long blobIndex = 3;
	FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures;
	
	holeCount = UnaryFeatures . GetBLOBHoleCount(blobList, blobImage, blobIndex);

Summary:	**Get the list of holes that are contained in a BLOB.**
Syntax:	CBlobList CUnaryFeature::GetBLOBHoleList (CBlobList blobList,

Summary:	**Get the list of holes that are contained in a BLOB.**
	CImage<blobImageClass, blobImgTyp> blobImage, unsigned long blobIndex);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobImage: This is the labeled image containing BLOBs. It should be a valid image and should not be empty. blobIndex: It represents the index into the list. Its value should be between 1 and size of blobList.
Return:	List of holes contained in a BLOB.
Description:	API finds the list of other labels contained in a BLOB.

Example:	CBlobList blobList;
	CBlobList holesList;
	CImage<CGrayImage16, GRAY16> blobImage;
	unsigned long blobIndex = 3;
	FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures;
	
	holesList = UnaryFeatures . GetBLOBHoleList(blobList, blobImage, blobIndex);

6.4 Required Classes for Binary Feature Extraction APIs

Class:	**CBinaryFeature Class**
Description:	This template class will expose various APIs that will extract various binary features from the labeled output image.
Syntax:	template <class blobImageClass, int blobImgTyp> class CBinaryFeature { public: static CDistance<long> GetBLOBDistance (CBlobList blobList, unsigned long blobIndex1, unsigned long blobIndex2); static char IsBLOBContained (CBlobList blobList, CImage<blobImageClass , blobImgTyp> blobImage, unsigned long blobIndex); static char IsBLOBTouching (CBlobList blobList, unsigned long blobIndex1, unsigned long blobIndex2,

Class:	CBinaryFeature Class

	neighborhood_type mode);
	
	static CBlobList SortBLOBs (CBlobList blobList, CImage<blobImageClass, blobImgTyp> blobImage, sort_type sortCriteria);
	}

Template Parameters	blobImageClass:-Input should be of Cimage class type e.g.: CGrayImage8, CGrayImage16, etc. blobImgTyp: Input should be of integer type e.g.: GRAY8, GRAY16, etc.
Member Functions:	**GetBLOBDistance:** This template function will find the minimum and maximum distance between two given BLOBs. **IsBLOBContained:** This template function will whether a given BLOB is contained in any other BLOB. **IsBLOBTouching**: This template function will whether a given pair of BLOBs are touching each other. **SortBLOB**: This template function will sort the blobs in the labeled image based on a set criteria.

6.5 Binary Feature Extraction APIs

1. CBinaryFeature::GetBLOBDistance()
2. CBinaryFeature::IsBLOBContained()
3. CBinaryFeature::IsBLOBTouching()
4. CBinaryFeature::SortBLOBs()

Summary:	**Compute the distance between two BLOBs.**
Syntax:	CDistance<long> CBinaryFeature::GetBLOBDistance (CBlobList blobList, unsigned long blobIndex1, unsigned long blobIndex2);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobIndex1: This represents the index into the list for the first BLOB. Its value should be between 1 and size of blobList. blobIndex2: This represents the index into the list for the second BLOB. Its value should be between 1 and size of blobList.
Return:	Distance between two given BLOBs.
Description:	The API finds the maximum and minimum distance between any pair of pixel one in each BLOB.
Example:	CBlobList blobList;

Summary:	**Compute the distance between two BLOBs.**

	CDistance<long> blobDistance;
	unsigned long blobIndex1 = 1;
	unsigned long blobIndex2 = 5;
	FOIL:: CBinaryFeature <CGrayImage16, GRAY16 > BinaryFeatures;
	
	blobDistance = BinaryFeatures . GetBLOBDistance(blobList, blobIndex1, blobIndex2);

Summary:	**Find whether a BLOB is contained in another.**
Syntax:	char CBinaryFeature::IsBLOBContained (CBlobList blobList, CImage<blobImageClass, blobImgTyp> blobImage, unsigned long blobIndex);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobImage: This is the labeled image containing BLOBs. It should be a valid image and should not be empty. blobIndex: This represents the index into the list. Its value should be between 1 and size of blobList.
Return:	Boolean, whether the given BLOB is contained in another or not.
Description:	The API finds whether a BLOB is contained in another or not.

Example:	CBlobList blobList;
	CImage<CGrayImage16, GRAY16> blobImage;
	unsigned long blobIndex = 3;
	char contained;
	FOIL:: CBinaryFeature <CGrayImage16, GRAY16 > BinaryFeatures;
	
	contained = BinaryFeatures . IsBLOBContained(blobList, blobImage, blobIndex);

Summary:	**Find whether BLOB's are touching.**
Syntax:	char CBinaryFeature::IsBLOBTouching (CBlobList blobList, unsigned long blobIndex1, unsigned long blobIndex2, neighborhood_type mode);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobIndex1: This represents the index into the list for the first BLOB. Its value should be between 1 and size of blobList. blobIndex2: This represents the index into the list for the second BLOB. Its value should be between 1 and size of blobList. Mode: Specifies consideration for touching based on 4- or 8-neighborhood.
Return:	Boolean, whether the given BLOB's are touching.

Summary:	**Find whether BLOB's are touching.**
Description:	The API finds the distance between the BLOB's and returns TRUE if distance is 0.

Example:	CBlobList blobList;
	unsigned long blobIndex1 = 1;
	unsigned long blobIndex2 = 5;
	FOIL:: CBinaryFeature <CGrayImage16, GRAY16 > BinaryFeatures;
	
	touching = BinaryFeatures . IsBLOBTouching(blobList, blobIndex1, blobIndex2, FOIL:: NEIGHBORHOOD8);

Summary:	**Sorts the BLOB's based on the set criteria.**
Syntax:	CBlobList CBinaryFeature::SortBLOBs (CBlobList blobList, CImage<blobImageClass, blobImgTyp> blobImage, sort_type sortCriteria);
Arguments:	blobList: This is the list of all BLOBs in a labeled image. The size of this list should not be zero and should contain valid values. blobImage: This is the labeled image containing BLOBs. It should be a valid image and should not be empty. sortCriteria: This is the criteria based on which the sorting is performed. The criteria range within the FOIL enumerated types of INCREASING_AREA, DECREASING_AREA, INCREASING_DIAMETER, DECREASING_DIAMETER, INCREASING_HOLECOUNT and DECREASING_HOLECOUNT .
Return:	Sorted list of BLOBs as per the set criteria.
Description:	API returns the sorted list of BLOB's as per the set criteria.

Example:	CBlobList blobList;
	CBlobList sortedList;
	CImage<CGrayImage16, GRAY16> blobImage;
	FOIL:: CBinaryFeature <CGrayImage16, GRAY16 > BinaryFeatures;
	
	sortedList = BinaryFeatures . SortBLOBs (blobList, blobImage, FOIL:: DECREASING_AREA);

7 PRACTICAL DEMONSTRATION OF THE FOIL APIS

7.1 Application of Pattern Matching Extension of FOIL Library

Consider a typical automated defect detection procedure. The user may want to verify whether the resistors plugged into a PCB are in correct slots. The defect detection setup will have an image capturing mechanism, which will be feeding in the snap shots of the manufactured PCB's.

There will be a reference image of the PCB with the resistors plugged into the correct slot. The user will determine the coordinate location of the resistors on the reference image of the PCB using the APIs supported in FOIL pattern recognition library. On getting the valid coordinate locations of the resistors the defect detection setup will go on applying the pattern recognition on the incoming snap shots of the PCB images, for location of the resistors. By comparing the coordinate location with the ones obtained from the reference image, we could determine the correctness of the slots into which the resistors are plugged in.

An application developer can solve the pattern-matching problem described above using one of the 3 available approaches supported in FOIL. He could use:

- Image based normalised correlation
- Image based difference
- Geometric based pattern matching

techniques based on the input and pattern images. If the input image and pattern images have a higher degree of correlation in terms of pixel contents the user will opt for either the Normalised correlation approach or the less accurate but faster image- based-differnce approach. If the pattern has well-defined geometric features then the user has the option of going with the geometric-based-pattern-matching approach.

What ever be the approach selected by the user, he will have to follow certain well-defined procedure in order to use FOIL to solve this problem.

FOIL supports a number of gray scale and color image formats. First and foremost the user will have to determine on the image types that constitute his problem.

For example: The input image type for the problem is Grayscale 8 bit images.

FOIL has a template-based class **CImage** that abstracts the different image types.

1.1.1.1 FOIL::CImage<FOIL::CGrayImage8, GRAY8> inputImage

The above template instantiation will convey that inputImage is a CGrayImage8 class, which is a FOIL class abstracting Gray scale 8-bit images. In addition to CImage class FOIL also supports color images as **CColourImage** template class. It takes the colour image class and its respective image type as template parameters. This can be instantiated as below.

FOIL::CColourImage<CRGBPlanar8, RGBPLANAR8> colourImage

In case of Image based correlation and Image based difference algorithms the user will have to provide a pattern image also.   FOIL::CImage<FOIL::CGrayImage8, GRAY8> patternImage

patternImage represents the pattern that is to be searched for in the input image. In case of the geometric-based approach, the user doesn't have to provide this patternImage. Instead, he will have to provide the point pattern representation of the pattern which will define the geometric property of the pattern to be searched.

In case where the user want to localize his pattern matching in both the patternImage and inputImage, he could provide respective masks to define the 'Region Of Interest'. If the whole images are of interest, then the masks should be all non-zero values (preferably 1). Otherwise, pixel locations of no interest will have pixel value '0'.

FOIL::CImage<FOIL::CGrayImage8, GRAY8> inputImageMask

FOIL::CImage<FOIL::CGrayImage8, GRAY8> patternMask

Once the image type is determined the user will have to build the model, which represents his problem space for pattern matching. This model will maintain the pattern in various rotational and scale hierarchy to achieve scale/rotational invariance while doing pattern matching.

The output generated by the ModelBuild api is a template based class called **CCombinedModel**. This template class will take the image class, image type and point type as the instantiation parameters. The point type is the type of the point that is defined as part of the point operator for geometric-based pattern matching. The general criterion is that the point type will have to be a data type that contains the range of the input image dimensions. It is advisable to define the point type as unsigned long since it usually holds the larger dimension.

FOIL::CCombinedModel<FOIL::CGrayImage8,GRAY8,unsigned long> resultModel

resultModel will hold the result of ModelBuild API that will be used subsequently by the Locate API.

The model build API will expect a pattern center of **CPoint** type in case of Image-based correlation or Image-based difference approach

FOIL::CPoint<unsigned long> patternCentre

In the case of Image-based correlation and Image-based difference APIs, we will need to call the ModelBuild API as below before calling the Locate API.

api_status = FOIL::ModelBuild (MODELIMAGECORRELATION, inputImage, patternImage, inputImageMask, patternMask, scaleStart, scaleStep, scaleEnd, angleStart, angleStep, angleEnd, resultModel, patternCentre)

parameter1: The type of approach for which the model is built. Only 2 valid selections are there for this parameter viz, MODELIMAGECORRELATION and MODLEIMAGEDIFFERENCE.

parameter2: input image as described in the above section
parameter3: pattern image as described in the above section
parameter4: input ROI mask as described in the above section
parameter5: pattern ROI mask as described in the above section

parameter6: the starting scale factor. This should be a number greater than or equal to 1.

paramter7: this is the step value upon which various subsequent scales are computed. The dimensions of the pattern image are multiplied by this factor subsequebtly to get new patterns of various sales.

parameter8: this is the end value of scale factor that is to be applied on the pattern image. It is advisable to limit the scale factors to the extent that the pattern image dimensions are always less than or equal to the input image dimensions.

parameter9: this is the start angle of the rotation. The angle can have a valid range between 0 and 360.

parameter10: this represents the angle step which will be used for maintaining patterns at various rotations.

parameter11: this represents the angle end

parameter12: this represents the object to catch the resultatnt model built by the api.

paramter13: this represents the Centre coordinates of the pattern. Normally the pattern center is (pattern_rows/2,pattern_col/2)

In the case of geometric-based pattern matching, we need not call the ModelBuild API. Only a preprocessing step is to be performed to add the base point patterns into the result model.

Assume the user defines three base-point pattern representations for the pattern that defines the complete geometry of the pattern. The user will be adding the base-point patterns to the result model as below.

CPatternElement is a template-based class that represents the point pattern definition of the geometric pattern to be searched. It has the image, image type and point type. The point type represents the data type to be used for the 'point.' If the user want to use Sobel as the point operator, he should use the **CPatternPointSobel** template class in place of CPatternElement.

FOIL:: CPatternElement<FOIL::CGrayImage8, GRAY8,int> basePattern1
FOIL:: CPatternElement<FOIL::CGrayImage8, GRAY8,int> basePattern2
FOIL:: CPatternPointSobel <FOIL::CGrayImage8, GRAY8,int> basePattern3
resultModel.Geometric.PatternList.push_back(basePattern1)
resultModel.Geometric.PatternList.push_back(basePattern2)
resultModel.Geometric.PatternList.push_back(basePattern3)

With the above set of procedures, the user has successfully built his model upon which he can apply the pattern location API. The Locate API will populate an object of **CMatchList** template-based class which will have the list of scores and match locations. The Locate API is invoked in the format shown below since a data type parameter that cannot be resolved prior to instantiation is required. The various parameters of this class are the image class, its associated type, the point type for the combined model which is dependent on the image dimensions, the data type for the match list locations which is dependent on the image dimensions, and the data type which hold the image pixel type.

api_status = FOIL::Locate< FOIL::CGrayImage8, GRAY8, unsigned long,unsigned long, unsigned char>(MODELIMAGECORRELATION, threshold, resultModel, match)

parameter1: : The type of pattern matching metric to be applied viz, MODELIMAGECORRELATION, MODLEIMAGEDIFFERENCE and MODELGEOMETRIC.

parameter2: This is the threshold to be specified across which the pattern matching results are screened. For image-based correlation, it is between 0 and 1.0. For geometric- based difference, it is a factor of the weight and value specified by the user in the base- point patterns. In case of image-based difference, it is driven by the input image and pattern image.

parameter3: This is the result of the ModelBuild API in the case of the image-based difference or correlation approach. It will be the preprocessed model with the base-point patterns populated for geometric-based approach.

parameter4: This is the result of the Locate API with scores and located coordinate points.

7.2 Application of Calibration/Measurement Extension of FOIL Library

Consider an application wherein real world objects are to be measured from digital images captured using a camera. The user will have to follow certain procedures using the FOIL library in order to achieve this goal.

The measuring procedure starts with a series of calibration procedures on the imaging setup. A specific imaging setup will require a number of corrections to be applied on its generated image in order to facilitate accurate measuring. FOIL provides a number of calibration APIs. The calibration APIs are member functions of a template-based class **CCalibratedImage**. This class needs to be instantiated before using any type of Calibration API. The template parameters required for instantiation are the image class, the corresponding image type and the image pixel data type. So if the calibration is to be performed on Grayscale 8 bit images, the user will instantiate the CCalibratedImage as follows:

FOIL::CCalibratedImage< CGrayImgae8, GRAY8,char> calibrationObj

If the calibration is to be done on a color image, the user should instantiate CCalibratedImage class as follows:

FOIL::CCalibratedImage< CRGBPlanar8, RGBPLANAR8,char> calibrationObj

The user could use any of the following APIs to perform calibration on the image to be measured. If the user want to perform dark field correction, then he could call the respective API with the respective inputs as shown below:

FOIL::CImage<CGrayImage8, GRAY8> outputImage
FOIL::CImage<CGrayImage8, GRAY8> inputImage
FOIL::CImage<CMatrixF, FLOAT> correctionFactor
*float** gain*
outputImage = calibrationObj . DarkFieldCorrection(inputImage, correctionFactor,gain)

parameter1: image to be corrected.
parameter2: correction image to be applied
parameter3: gain factor to be applied to each pixel location

The API will return the corrected image.

If the user needs to perform calibration on color images, then he will instantiate the **CColourImage** template-based class for input and output calling the overloaded dark field correction algorithm. An example os this type of calibration on a color image is:

FOIL::CColourImage<CRGBPlanar8, RGBPLANAR8> outputImage
FOIL::CColourImage< CRGBPlanar8, RGBPLANAR8> inputImage
FOIL::CImage<CMatrixF, FLOAT> RedFactor
FOIL::CImage<CMatrixF, FLOAT> GreenFactor
FOIL::CImage<CMatrixF, FLOAT> BlueFactor
float** redGain
float** greenGain
float** blueGain

outputImage = calibrationObj . DarkFieldCorrection(inputImage, RedFactor, GreenFactor, BlueFactor, redGain, greenGain, blueGain)

parameter1: the image to be corrected
parameter2: the correction image for RED channel
parameter3: the correction factor for GREEN channel
parameter4: the correction factor for BLUE channel

parameter5: the gain to be applied on each pixel in RED channel
parameter6: the gain to be applied on each pixel in GREEN channel
parameter7: the gain to be applied on each pixel in BLUE channel
This API will return the corrected image.

If the user wants to perform Chromatic aberration correction, he could call the following API.

FOIL::CColourImage<CRGBPlanar8, RGBPLANAR8> outputImage
FOIL::CColourImage< CRGBPlanar8, RGBPLANAR8> inputImage
double Rscale
double Gscale
double Bscale
double centreX
double centreY

outputImage = calibrationObj . ColourAberrationCorrection (inputImage, Rscale, Gscale, Bscale,centreX, centreY)

parameter1: the input image to be corrected
parameter2: scale factor to be applied on the RED channel
parameter3: scale factor to be applied on the GREEN channel
parameter4: scale factor to be applied on the BLUE channel
parameter5: the Xcenter of the scale.
parameter6: the Ycenter of the scale
This API will return the corrected image

If user wants to perform scale correction, he could use the following:

FOIL::CImage<CGrayImage8, GRAY8> outputImage
FOIL::CImage<CGrayImage8, GRAY8> inputImage
float XScale
float YScale
float XShear
float YShear
float XTrans
float YTrans

outputImage = calibrationObj . ScaleCorrection (inputImage, XScale, YScale, XShear, YShear, XTrans, YTrans)
parameter1: the image to be corrected.
parameter2: scale factor on X direction
parameter3: scale factor on Y direction
parameter4: shear factor on X direction
parameter5: translation factor in X direction
parameter6: translation factor in Y direction
This API will return the corrected image.

If user wants to perform perspective correction, he should use the following:

FOIL::CImage<CGrayImage8, GRAY8> outputImage
FOIL::CImage<CGrayImage8, GRAY8> inputImage
FOIL:: CVector<long> u_vector
FOIL:: CVector<long> v_vector
FOIL:: CVector<long> n_vector
float L

outputImage = calibrationObj . PerspectiveCorrection (inputImage, u_vector, v_vector, n_vector, L)

parameter1: image to be corrected
parameter2: The u transformation vector to be applied
parameter3: the v transformation vector to be applied
parameter4: the n transformation vector to be applied
parameter5: the sensor specification
This API will return the corrected image.

In order to make the measurements in real world metrics, there should be a method to translate the image measurements to real measurements. For this, the user will use an array-of-dots image of known distance. He will be subjecting this image to all the corrections to calibrate his input image. After performing correction on the array-of-dots image, FOIL provides the user with a set of APIs which will help him to translate image measurements to real measurements. The user will instantiate an object of **CCoordinateSystem** template-based class. The parameters of this template class are the number of holes in the row (the convention is that the array-of-dots image will be constituted on an NxN matrix of filled holes which are equidistant in both X and Y directions), Image class, image type and the point type that is to be passed for the geometric point pattern representation of a hole. This API uses the geometric-pattern matching to locate the holes in the array-of-dots image and define the translation factor accordingly. For a 10x10 array-of-dots grayscale 8-bit image, the user will instantiate the CCoordinateSystem class as follows:

FOIL::CCoordinateSystem<10,char, CGrayImage8, GRAY8> ordinate

FOIL provides the user with the following APIs to enable him to translate the image measurements to real world measurements. FOIL defines a SetMeasurementCoordinates API for this:

FOIL::CImage<CGrayImage8, GRAY8> arrayOfDots10by10Image
FOIL::CImage<CGrayImage8, GRAY8> pointPattern
float threshold = 1

ordinate . SetMeasurementCoordinates(arrayOfDots10by10Image, pointPattern, threshold)

FOIL provides the user with GetUnitInPixels to get the translation factor to be applied to extract real world measurements from image measurement units.

int dotLength
dotLength = ordinate . GetUnitInPixels(void)

This API will return the translation factor to be used to convert the image measurements to real world measurements.

The user may want to find the orientation of the object from the input image which contains the object to be measured. FOIL assumes a single object is present in the input image to be oriented. FOIL provides four APIs that will help the user in determining the orientation of the object. In order to use the orientation algorithms, the user will have to instantiate an object of the **COrient** template-based class. The template parameters are the position type which defines the data type to hold the top-left coordinate locations of the bounding box rectangle object, and the size type which defines the data type to hold the type of the object holding length and height of the bounding box rectangle object. Both are dependent on the image dimensions. The remaining parameters to the template are the image class, its respective image type and the data type of the pattern point for the filled hole to be searched in the array-of-dots image.

FOIL::COrient<int,int, CGrayImage8, GRAY8,int> orientObj

The various APIs provided by FOIL to assist the user in determining the orientation of the object are as follows. The user can use the GetBoundinBox API to get the minimum bounding box that contains the object.

FOIL::CImage<CGrayImage8, GRAY8> objectImage
FOIL::CRectangle<int,int> minBoundingBox
minBoundingBox = orientObj . GetBoundingBox (objectImage) parameter1: The input image that has the single object whose orientation is to be found.

This API will return the minimum bounding box that can hold the image.

Based on the geometry of the object, the user may not be able to find its orientation just by getting the minimum bounding box. In that case, the user could use the FindHole API to locate a hole in the object to determine the orientation. This API should be called after calling the GetBoundingBox API.

```
FOIL::CImage<CGrayImage8, GRAY8> objectImage
FOIL:: CPatternElement<int, CGrayImage8, GRAY8> holePointPattern
FOIL::CRectangle<int,int> minBoundingBox
float threshold = 1
FOIL::HolesList holes
```

return = orientObj . FindHole (objectImage, holePointPattern, minBoundingBox, threshold, holes)

parameter1: input image containing the single object whose orientation is to be determined

parameter2: point pattern representation of the filled hole.
parameter3: the bounding box within which the hole is to be located.
parameter4: the threshold to be applied locating the hole.

parameter5: the holes list returned by the api which contains the location where all the holes are found.

FOIL also provides another API, FindLine, to find the orientation of the object. This API also should be called after calling the GetBoundingBox API. This API will find the location of a line segment of the specified length on the object boundary.

```
FOIL::CImage<CGrayImage8, GRAY8> objectImage
FOIL::CRectangle<int,int> minBoundingBox
FOIL::CLineSegment<long> line
long length = 25
```

return = orientObj . FindLine (objectImage, length, minBoundingBox, line)

parameter1: the image containing the single object to be oriented
parameter2: the length of the line to be located

parameter3: this specifies the bounding box area in the image where the location of the line is to be limited to.

parameter4: this is coordinates of the located line segment.

FOIL provides another API, FindCorner, to determine the orientation. This API also should be called after GetBoundingBox API.

```
FOIL::CImage<CGrayImage8, GRAY8> objectImage
  FOIL::CRectangle<int,int> minBoundingBox
  FOIL::CornerPairs corners

    return = orientObj . FindCorner (objectImage, minBoundingBox, corners)
```

parameter1: the input image which contains the object whose orientation to be found.

parameter2: the boundinx box area inside the image where the location for linesegments with a common vertex is to be limited.

parameter3: The result of the corner located which is the coordinates of the 2 lines segments having a common vertex.

Once the user has determined the orientation of the object whose features are to be measured, he could define his measurement tools on the image from which the measurement is to be extracted. If the user needs to find the length of an edge of the object, given the computed orientation of that object edge he coulde measurement tools perpendicular to the edge direction to get accurate measurements. FOIL provides the user with a template-based class to define the measurement tool called **CMeasure**. The user will have to instantiate the CMeasure object in order to use the Measure and other Fit APIs. This template class accepts classes of type image class, image type, point type, (depending on the image dimensions) and image data type that is the type to hold the image type.

```
    FOIL::CMeasure< CGrayImage8, GRAY8,long,char> measuringObj
```

Once the measuring object is instantiated, the user could use the various methods to extract the point sets from the defined measurement tool and fit different geometries to the extracted points. FOIL provides the Measure API that will extract the point set using a single pixel edge detection with the measurement tool defined by the user. The user orients the measurement tool based on the orientation of the object.

```
  FOIL::CMeasurementTool<long> toolset
  FOIL::CPointSet<long> points
    FOIL::CImage<CGrayImage8, GRAY8> objectImage
    points = measuringObj . Measure (toolset, objectImage)
```

paramter1: the measurement tool set comprising of a series of linesegments across which the single pixel edge detection is to be performed.

parameter2: the image that contains the object whose feature is to be measured.

This API will return the point set resultatnt from the edge detection run over the user-defined toolset.

Once the user has detected the point set across his measurement tool, he can use various Fit algorithms to fit the geometry on the point set whose features can be extracted. FOIL provides the overloaded Fit API for fitting a line and circle. This API should be called after calling the Measure API. The following demonstrates the FOIL Fit API call to fit a line into a give point set.

```
FOIL::CPointSet<long> points
        FOIL::CLineSegment<long> FittedLine
        return = measuringObj . Fit (points, FittedLine)
```

parameter1: the point set extracted by the Measure API call into which a line is to be fitted

parameter2: the line fitted into the point set whose features could be extracted.

The following demonstrates the FOIL Fit API call to fit a circle into the given point set.

FOIL::CPointSet<long> points
FOIL::CCircle<long> FittedCircle

return = measuringObj . Fit (points, FittedCircle)

parameter1: the point set extracted by the Measure API call into which a circle is to be fitted

parameter2: the circle fitted into the point set whose features could be extracted

FOIL also provides an API that will perform the single pixel edge detection along the specified line segment on the image. The Measure API uses the DetectEdge API internally to get the point sets.

FOIL::CPointSet<long> point
FOIL::CLineSegment<long> lineSegment
FOIL::CImage< CGrayImage8, GRAY8> image

point = measuringObj .DetectEdge(lineSegment, image)
parameter1: the line segment along which the edge detection is to be performed.
parameter2: the image upon which the edge detection is to be performed.
This API will return the detected edge point along the line segment.

Once the user has fit definite geometries (line or circle) into the collected point set, FOIL provides a set of Spatial and Photometric APIs to the user to make actual measurement of features. The user should instantiate an object of **CSpatialMetricTools** template class. This template takes the point type which is the data type to hold the coordinate points which is dependent on the image dimensions, the row holes which is same as the number of holes in a row in the array-of-dots image, the image class and its corresponding image type.

FOIL::CSpatialMetricTools<10, CGrayImage8, GRAY8> SpatialTool

If the user needs to extract the length between 2 end points of a line segment in the X direction, FOIL provides the XLength API.

FOIL::CLineSegment<int> line

FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10
length = SpatialTool . XLength (line, ordinate10by10)
parameter1: the fitted line segment from the point set extracted by the Measure

API.

parameter2: the coordinate system which is to be used for translating the image measurement into real world meaurement.
This API will return the length of a line segment along the X axis.

If the user needs to extract the length between 2 end points of a line segment in the Y direction, FOIL provides the YLength API.

FOIL::CLineSegment<int> line

FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10
length = SpatialTool . YLength (line, ordinate10by10)

parameter1: the fitted line segment from the point set extracted by the Measure api.

parameter2: the coordinate system which is to be used for translating the image measurement into real world meaurement.

This API will return the length of a line segment along the Y axis.

If the user needs to extract the absolute length a line segment, FOIL provides the Length API.

FOIL::CLineSegment<int> line

FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10
length = SpatialTool . Length (line, ordinate10by10)
parameter1: the fitted line segment from the point set extracted by the Measure API.
parameter2: the coordinate system which is to be used for translating the image measuremenst into real world meaurements.

This API will return the absolute length of a line segment.

If the user needs to extract angle made by a line segment with X-axis, FOIL provides the XAngle API.

FOIL::CLineSegment<int> line

angle = SpatialTool . XAngle (line);
parameter1: the fitted line segment from the point set extracted by the Measure api.

This API will return the angle in degrees made by the line segment with X axis.

If the user needs to extract angle made by a line segment with Y-axis, FOIL provides the YAngle API.

FOIL::CLineSegment<int> line
angle = SpatialTool . YAngle (line);
parameter1: the fitted line segment from the point set extracted by the Measure api.

This API will return the angle in degrees made by the line segment with Y axis.

If the user needs to extract angle made by a line segment with X-axis, FOIL provides the XAngle API.

FOIL::CLineSegment<int> line

angle = SpatialTool . XAngle (line);

parameter1: the fitted line segment from the point set extracted by the Measure api.

This API will return the angle in degrees made by the line segment with X axis.

If the user needs to extract angle made by a line segment with the other, FOIL provides the Angle API.

FOIL::CLineSegment<int> line1
FOIL::CLineSegment<int> line2

angle = SpatialTool . Angle (line1,line2);

parameter1: the fitted line segment from the point set extracted by the Measure api.

parameter2: the fitted line segment from the point set extracted by the Measure api.

This API will return the angle in degrees made by the line segment1 with line segment2.

If the user want to find the angle inscribed in an arc, FOIL provides him the overloaded version of Angle API.

FOIL::CArc<int> arc

angle = SpatialTool . Angle (arc) parameter1: this is the arc whose inclusive angle is to be measured.

This API will return the inscribing angle made by the radial line segment of the arc at its end points.

If user wants to determine the diameter of the circle fitted by the Measure api, FOIL provides the Diameter API.

FOIL::CCircle<int> circle

FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10
diameter = SpatialTool . Diameter (circle, ordinate10by10)

parameter1: the circle fitted by the Measure api whose diameter is to be computed in real world units.

parameter2: the coordinate system which is to be used for translating the image measurement into real world measurement.

This API will return the diameter of the circle in real world units.

If user wants to determine the radius of the circle fitted by the Measure API, FOIL provides the Radius API.

FOIL::CCircle<int> circle

FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10
radius = SpatialTool . Radius (circle, ordinate10by10)
parameter1: the circle fitted by the Measure api whose radius is to be computed in real world units.
parameter2: the coordinate system which is to be used for translating the image measurements into real world measurements.

This API will return the radius of the circle in real world units.

If user wants to find out the area on an image specified by a region, FOIL provides the Area API.

FOIL::CRegion<int> searchArea

FOIL::CCoordinateSystem<10,int, CGrayImage8, GRAY8> ordinate10by10
area = SpatialTool . Area (searchArea, ordinate10by10) parameter1: the region whose area is to be computed in real world units. parameter2: the coordinate system which is to be used for translating the image measurements into real world measurements.

This API will return the area of the region in real world units.

In addition to spatial metric tools, FOIL also provides the user with a set of Photometric measurement APIs. The user will have to instantiate an object of **CPhotometricTools** template-based class. The parameters of this template class are the color type that denotes the data type to hold the pixel intensity value which is dependend on the image type, the point type that defines the type to contain the image region coordinates which are dependent on the image dimensions, the image class and its respective image type.

FOIL::CPhotoMetricTools<int, int, CRGBPlanar8, RGBPLANAR8> PhotometricTool

FOIL supply two photometric measurement APIs. GetAverageColour is the API that will get the average intensity of each color channel in the specified region of the image. This API is meant for color scale images alone.

FOIL::CRegion<int> searchArea
FOIL::CColour<int> AverageColour
FOIL::CColourImage< CRGBPlanar8, RGBPLANAR8> inputImage

AverageColour = PhotometricTool . GetAverageColour (searchArea, inputImage)
parameter1: this is the area on the input image where the average colour intensity is to be computed.
parameter2: This is the input image upon which the intensity value is computed.

This API will return the average colour intensity for each channel.

If the user wants to find out the average gray scale value of an image in a specified region, FOIL provides the GetAverageGray API. This API is meant for gray scale images alone.

FOIL::CRegion<int> searchArea;
FOIL::CImage< CGrayImage8, GRAY8> inputImage;
averageGray = PhotometricTool . GetAverageGray (searchArea, inputImage)
parameter1: this is the area on the input image where the average gray scale is to be computed
parameter2: This is the input image upon which the gray value is computed.

This API will return the average gray scale in the image on the specified region.

With the above set of measurement APIs, FOIL enables the user to perform various feature measurements that can be used in real applications.

7.3 Application of Morphology/BLOB Analysis Extension of FOIL Library

Consider a typical application where the user wants to automate the process of computing the number of specifically-shaped pills manufactured in a pharmaceutical production process. The imaging system will provide snap shots of each sector of the produced pills on the conveyor belt. The user will have to make an application that will count for the number of pills manufactured with a specific diameter. Here the diameter of a pill is the distinguishing factor. FOIL gives the morphology and BLOB analysis extension to approach this problem.

Morphology extension of FOIL is used as a preprocessing step to the BLOB analysis phase that actually enables the user to extract the BLOB features. FOIL provides a set of Morphology APIs that will prepare the images to be labeled in the subsequent BLOB analysis phase. The user will have to instantiate a **CBinarize** class to get access to the morphology APIs related with thresholding. This is a template-based class that takes the input image class and its associated type, the threshold image class and its associated type and the output image class and its associated type as template parameters. The image types are Gray scale images. It is always preferable to have the background pixel put a s '0' as some of the following logic will expect the background pixel to be zero.

FOIL::CBinarize< CGrayImage16, GRAY16, CGrayImage8, GRAY8, CGrayImage8, GRAY8> Thresholding

FOIL provides two APIs to perform Thresholding. FOIL support the FixedThreshold API as one version of this.

```
FOIL::CImage<CGrayImage8, GRAY8> inputGrayImage
FOIL::CImage<CGrayImage8, GRAY8> * BinarizedImage
int  lowThreshold = 40
int highThreshold = 70
int  foregroundPixel = 100
int backgroundPixel = 0
```

status = Thresholding . FixedThreshold(inputGrayImage, lowThreshold , highThreshold, FOIL::RANGE_OUTSIDE_THRESHOLD, foregroundPixel, backgroundPixel, BinarizedImage)

parameter1: the gray scale input image on which thresholding is to be done.
parameter2: the lower value of threshold.
parameter3: the upper value of threshold.
parameter4: specification whether the range is to checked is within or beyond the specified thresholds.
parameter5: the pixel value to be put in the foreground of the binarized image.
parameter6: the pixel value to be put in the background of the binarized image.
parameter7: the resultant image of thresholding.

FOIL also support the dynamic thresholding for binarising the images. The user can use the same interface to return thresholding image as either 8-bit gray scale or 16-bit gray scale.

```
FOIL::CImage<CGrayImage16, GRAY16> inputGrayImage
FOIL::CImage<CGrayImage8, GRAY8> * BinarizedImage
FOIL::CImage< CGrayImage16, GRAY16>  lowThreshold
FOIL::CImage< CGrayImage16, GRAY16> highThreshold
int  foregroundPixel = 100
int backgroundPixel = 0
```

status = Thresholding . DynamicThreshold(inputGrayImage, lowThreshold , highThreshold, FOIL:: RANGE_WITHIN_THRESHOLD, foregroundPixle, backgroundPixle, BinarizedImage)

parameter1: the gray scale input image on which thresholding is to be done.
parameter2: the image having lower value of threshold across each pixel.
parameter3: the image having upper value of threshold across each pixel.

parameter4: specification whether the range is to checked is within or beyond the specified thresholds.
parameter5: the pixel value to be put in the foreground of the binarized image.
parameter6: the pixel value to be put in the background of the binarized image.
parameter7: the resultant image of thresholding

Once the user has binarized the input image, then FOIL supplies him with a number of structuring element APIs that will condition the binarized image for labeling in BLOB analysis phase. The user will have to instantiate an object of **CStructureElement** template-based class. This template class takes the image class and its respective type as the parameters.

```
FOIL::CStructureElement< CGrayImage8, GRAY8> Structuring
```

The various structuring APIs provided by FOIL are detailed below. The user can use the Erosion API for structuring his binarized image:

```
FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage
FOIL::CImage<CGrayImage8, GRAY8> * ErodedImage
int  foregroundPixel = 100
int backgroundPixel = 0
```

status = Structuring . Erosion(BinarizedImage, FOIL:: TEMPLATE3x3, foregroundPixel, backgroundPixle, ErodedImage)

parameter1: the binarized input image that is to be eroded.
parameter2: the template type to be used for erosion.
parameter3: the pixle value to be put in foreground.
parameter4: the pixel value to be put in the background
parameter5: the resultant image of erosion.

If the user want to dilate the binarized image then he could use the Dilation API in FOIL.

```
FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage
FOIL::CImage<CGrayImage8, GRAY8> * DilatedImage
int  foregroundPixel = 100
int backgroundPixel = 0
```

status = Structuring . Dilation(BinarizedImage, FOIL:: TEMPLATE5x5, foregroundPixel, backgroundPixle, DilatedImage)

parameter1: the binarized input image that is to be dilated.
parameter2: the template type to be used for dilation.
parameter3: the pixle value to be put in foreground.
parameter4: the pixel value to be put in the background
parameter5: the resultant image of erosion.

If the user want to perform opening on the binarized image, FOIL provides him with the Open API.

```
FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage
FOIL::CImage<CGrayImage8, GRAY8> * OpenedImage
int repetitionFactor = 5
int  foregroundPixel = 100
int backgroundPixel = 0
```

status = Structuring . Open(BinarizedImage, FOIL:: TEMPLATE7x7, repetitionFactor, foregroundPixel, backgroundPixle, OpenedImage)

parameter1: the binarized input image that is to be opened.
parameter2: the template type to be used for opening.
parameter3: the number of times opening is to be performed.
parameter4: the pixle value to be put in foreground.
parameter5: the pixel value to be put in the background
parameter6: the resultant image of opening

If the user want to perform closing on the binarized image, FOIL provides him with the Close API.

```
FOIL::CImage<CGrayImage8, GRAY8> BinarizedImage
FOIL::CImage<CGrayImage8, GRAY8> * ClosedImage
int repetitionFactor = 5
int  foregroundPixel = 100
int backgroundPixel = 0
```

status = Structuring . Close(BinarizedImage, FOIL:: TEMPLATE3x3, repetitionFactor, foregroundPixel, backgroundPixle, ClosedImage)

parameter1: the binarized input image that is to be closed.
parameter2: the template type to be used for closing.
parameter3: the number of times closing is to be performed.
parameter4: the pixle value to be put in foreground.
parameter5: the pixel value to be put in the background
parameter6: the resultant image of closing

After the conditioning of the gray scale image is done using the Morphology APIs the user is ready to start the BLOB analysis phase. The primary step of doing BLOB analysis is to label the conditioned binarized image into distinct BLOBs. FOIL provides LabelBLOBs API for achieving this goal. This is a template-based function which can be instantiated with required input parameters.

```
FOIL::CImage<CGrayImage8, GRAY8> inputImage
FOIL::CImage<CgrayImage16, GRAY16> * blobImage
CBlobList blobList
int foregroundPixel = 100
```

err_ret = FOIL::LabelBLOBs(inputImage, FOIL:: NEIGHBORHOOD4, foregroundPixel , blobImage, blobList)

paramter1: this is the conditioned binarized image to be labeled.
parameter2: the neighborhood type to be used for defining the BLOB touching.
parameter3: the labeled image output.

parameter4: the blob list object containing the list of all points in all BLOBs respectively.

Once the user has successfully labeled the image then he can use either the unary feature APIs or binary feature APIs to extract the BLOB characteristics. The user will have to instantiate an object of **CUnaryFeature** template-based class. The template-based class will take blob image class and respective image type as the parameters.

FOIL::CUnaryFeature<CGrayImage16, GRAY16> UnaryFeatures

If the user wants to extract the centroid of a specified blob he can use the GetBLOBCentroid API provided in FOIL.

CBlobList blobList
unsigned long blobIndex = 3
CPoint<long> centroid

centroid = UnaryFeatures . GetBLOBCentroid(blobList, blobIndex) parameter1: the blob list generated by the previous labeling operation. parameter2: the 1 based index for the BLOB whose feature is to be determined.
This API will return the centroid of the specified BLOB.

If the user wants to extract the perimeter of a specified blob he can use the GetBLOBPerimeter API provided in FOIL.

```
CBlobList blobList
CImage<CgrayImage16, GRAY16> blobImage
CBlobList perimeterList
unsigned long blobIndex = 3;
```

perimeterList = UnaryFeatures . GetBLOBPerimeter(blobList, blobImage , blobIndex)

parameter1: the blob list generated by the previous labeling operation.
parameter2: the blob image generated by the previous labeling operation.
parameter3: the 1 based index for the BLOB whose feature is to be determined.

This API will return the perimeter of the specified BLOB.

If the user wants to extract the area of a specified blob he can use the GetBLOBArea API provided in FOIL.

```
CBlobList blobList
unsigned long blobIndex = 3
long blobArea;
```

blobArea = UnaryFeatures . GetBLOBArea(blobList, blobIndex)
parameter1: the blob list generated by the previous labeling operation.
parameter2: the 1 based index for the BLOB whose feature is to be determined.

This API will return the area of the specified BLOB.

If the user wants to extract the binary image of a specified blob he can use the ExtractBLOB API provided in FOIL.

```
CBlobList blobList
CImage<CGrayImage16, GRAY16> blobImage
CImage<CGrayImage16, GRAY16> * extractedBLOB
char foreground Pixel = 1
char backgroundPixel = 0
unsigned long blobIndex = 3
```

status = UnaryFeatures . ExtractBLOB(blobList, blobImage, blobIndex, foregroundPixel, backgroundPixel, extractedBLOB)

parameter1: the blob list generated by the previous labeling operation.
parameter2: the blob image generated by the previous labeling operation.
parameter3: the 1 based index for the BLOB whose feature is to be determined.
parameter4: the pixel value to be put in foreground.
parameter5: the pixel value to be put in background.

parameter6: the binary image containing the extracted BLOB alone.

If the user wants to extract the smallest bounding box containing the specified blob he can use the GetBLOBBoundingBox API provided in FOIL.

CBlobList blobList
CImage<long> boundingBox
unsigned long blobIndex = 3

boundingBox = UnaryFeatures . GetBLOBBoundingBox(blobList, blobIndex)
parameter1: the blob list generated by the previous labeling operation.
parameter2: the 1 based index for the BLOB whose feature is to be determined.

This API will return the minimum bounding box that can contain the specified

BLOB.

If the user wants to extract the diameter of a specified blob he can use the GetBLOBDiameter API provided in FOIL.

CBlobList blobList
float blobDiameter
long blobIndex = 3

blobDiameter = UnaryFeatures . GetBLOBDiameter(blobList, blobIndex)
parameter1: the blob list generated by the previous labeling operation.
parameter2: the 1 based index for the BLOB whose feature is to be determined.

This API will return the diameter of the specified BLOB.

If the user wants to extract the number of holes in a specified blob he can use the GetBLOBHoleCount API provided in FOIL.

CBlobList blobList
CImage<CGrayImage16, GRAY16> blobImage
long holeCount

holeCount = UnaryFeatures . GetBLOBHoleCount(blobList, blobImage, blobIndex)

parameter1: the blob list generated by the previous labeling operation.
parameter2: the blob image generated by the previous labeling operation.
parameter3: the 1 based index for the BLOB whose feature is to be determined.
This API will return the number of holes contained in the specified BLOB.

If the user wants to extract the list of holes in a specified blob he can use the GetBLOBHoleList API provided in FOIL.

CBlobList blobList
CBlobList holesList
CImage<CGrayImage16, GRAY16> blobImage
unsigned long blobIndex = 3

holesList = UnaryFeatures . GetBLOBHoleList(blobList, blobImage, blobIndex);

parameter1: the blob list generated by the previous labeling operation.
parameter2: the blob image generated by the previous labeling operation.
parameter3: the 1 based index for the BLOB whose feature is to be determined.
This API will return the list of all holes contained in the specified BLOB.

In addition to the unary features depicted above, FOIL also provides some APIs that extract the binary features of BLOBs. User will have to instantiate an object of **CBinaryFeature** template class. This template takes the blob image class and its respective type as the parameters.

FOIL:: CBinaryFeature <CGrayImage16, GRAY16 > BinaryFeatures

The FOIL supports a number of binary features. If the user wants to determine the distance between 2 BLOBs, FOIL provides the user with GetBLOBDistance API.

```
CBlobList blobList
CDistance<long> blobDistance
unsigned long blobIndex1 = 1
unsigned long blobIndex2 = 5
```

blobDistance = BinaryFeatures . GetBLOBDistance(blobList, blobIndex1, blobIndex2)

parameter1: the blob list generated by the previous labeling operation.

parameter2: the 1 based index for one of the BLOB whose feature is to be determined.

parameter2: the 1 based index for one of the BLOB whose feature is to be determined.

This API will return the distance between the 2 specified BLOBs.

If the user wants to determine whether a BLOB is contained in another, FOIL provides the user with IsBLOBContained API.

```
CBlobList blobList
CImage<CGrayImage16, GRAY16> blobImage
unsigned long blobIndex = 3
char contained
```

contained = BinaryFeatures . IsBLOBContained(blobList, blobImage, blobIndex)
parameter1: the blob list generated by the previous labeling operation.
parameter2: the blob image generated by the previous labeling operation.

parameter3: the 1 based index for one of the BLOB whose feature is to be determined.

This API will return whether a BLOB is contained in another or not.

If the user wants to determine whether a BLOB is touching another, FOIL provides the user with IsBLOBTouching api.

```
CBlobList blobList
unsigned long blobIndex1 = 1
unsigned long blobIndex2 = 5
```

touching = BinaryFeatures . IsBLOBTouching(blobList, blobIndex1, blobIndex2, FOIL:: NEIGHBORHOOD8);

parameter1: the blob list generated by the previous labeling operation.

parameter2: the 1 based index for one of the BLOB whose feature is to be determined.

parameter3: the 1 based index for one of the BLOB whose feature is to be determined.

parameter4: the type of neighborhood to be used to define BLOB touching.

This API will return whether a BLOB is touching another or not.

For the purpose of sorting the blobs in the blob list based on different criteria, FOIL provides the user with SortBLOBs API.

```
CBlobList blobList
CBlobList sortedList
CImage<CGrayImage16, GRAY16> blobImage;
```

sortedList = BinaryFeatures . SortBLOBs (blobList, blobImage, FOIL:: DECREASING_AREA);

parameter1: the blob list generated by the previous labeling operation.
parameter2: the blob image generated by the previous labeling operation.
parameter3: the sort criteria to be applied.

This API will return the blob list sorted based on the specified criteria.

8 TROUBLESHOOTING

There are several things you can try before you call Alacron Technical Support for help.

_____ Make sure the computer is plugged in. Make sure the power source is on.

_____ Go back over the hardware installation to make sure you didn't miss a page or a section.

_____ Go back over the software installation to make sure you have installed all necessary software.

_____ Run the Installation User Test to verify correct installation of both hardware and software.

_____ Run the user-diagnostics test for your main board to make sure it's working properly.

_____ Insert the Alacron CD-ROM and check the various Release Notes to see if there is any information relevant to the problem you are experiencing.

The release notes are available in the directory: \usr\alacron\alinfo

_____ Compile and run the example programs found in the directory: \usr\alacron\src\examples

_____ Find the appropriate section of the Programmer's Guide & Reference or the Library User's Manual for the particular library and problem you are experiencing. Go back over the steps in the guide.

_____ Check the programming examples supplied with the runtime software to see if you are using the software according to the examples.

_____ Review the return status from functions and any input arguments.

_____ Simplify the program as much as possible until you can isolate the problem. Turning off any operations not directly related may help isolate the problem.

_____ Finally, first save your original work. Then remove any extraneous code that doesn't directly contribute to the problem or failure.

9 ALACRON TECHNICAL SUPPORT

Alacron offers technical support to any licensed user during the normal business hours of 9 a.m. to 5 p.m. EST. We offer assistance on all aspects of processor board and PMC installation and operation.

9.1 Contacting Technical Support

To speak with a Technical Support Representative on the telephone, call the number below and ask for Technical Support:

Telephone: **603-891-2750**

If you would rather FAX a written description of the problem, make sure you address the FAX to Technical Support and send it to:

Fax: **603-891-2745**

You can email a description of the problem to ***support@alacron.com***

Before you contact technical support have the following information ready:

_____ Serial numbers and hardware revision numbers of all of your boards. This information is written on the invoice that was shipped with your products.

_____ Also, each board has its serial number and revision number written on either in ink or in bar-code form.

_____ The version of the ALRT, ALFAST, or FASTLIB software that you are using.

_____ You can find this information in a file in the directory: **\usr\alfast\alinfo**

_____ The type and version of the host operating system, i.e., Windows 98.

_____ Note the types and numbers of all your software revisions, daughter card libraries, the application library and the compiler

_____ The piece of code that exhibits the problem, if applicable. If you email Alacron the piece of code, our Technical-Support team can try to reproduce the error. It is necessary, though, for all the information listed above to be included, so Technical Support can duplicate your hardware and system environment.

9.2 Returning Products for Repair or Replacements

Our first concern is that you be pleased with your Alacron products.

If, after trying everything you can do yourself, and after contacting Alacron Technical Support, you feel your hardware or software is not functioning properly, you can return the product to Alacron for service or replacement. Service or replacement may be covered by your warranty, depending upon your warranty.The first step is to call Alacron and request a "Return Materials Authorization" (RMA) number.This is the number assigned both to your returning product and to all records of your communications with Technical Support. When an Alacron technician receives your returned hardware or software he will match its RMA number to the on-file information you have given us, so he can solve the problem you've cited.

When calling for an RMA number, please have the following information ready:

_____ Serial numbers and descriptions of product(s) being shipped back

_____ A listing including revision numbers for all software, libraries, applications, daughter cards, etc.

_____ A clear and detailed description of the problem and when it occurs

_____ Exact code that will cause the failure

_____ A description of any environmental condition that can cause the problem

All of this information will be logged into the RMA report so it's there for the technician when your product arrives at Alacron.Put boards inside their anti-static protective bags. Then pack the product(s) securely in the original shipping materials, if possible, and ship to:

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Clearly mark ***the outside of your package:***

Attention **RMA #80XXX**

Remember to include your return address and the name and number of the person who should be contacted if we have questions.

9.3 Reporting Bugs

We at Alacron are continually improving our products to ensure the success of your projects. In addition to ongoing improvements, every Alacron product is put through extensive and varied testing. Even so, occasionally situations can come up in the fields that were not encountered during our testing at Alacron.

If you encounter a software or hardware problem or anomaly, please contact us immediately for assistance. If a fix is not available right away, often we can devise a work-around that allows you to move forward with your project while we continue to work on the problem you've encountered.

It is important that we are able to reproduce your error in an isolated test case. You can help if you create a stand-alone code module that is isolated from your application and yet clearly demonstrates the anomaly or flaw.

Describe the error that occurs with the particular code module and email the file to us at:

support@alacron.com

We will compile and run the module to track down the anomaly you've found.

If you do not have Internet access, or if it is inconvenient for you to get to access, copy the code to a disk, describe the error, and mail the disk to Technical Support at the Alacron address below.

If the code is small enough, you can also:

FAX the code module to us at 603-891-2745

If you are faxing the code, write everything large and legibly and remember to include your description of the error.

When you are describing a software problem, include revision numbers of all associated software.

For documentation errors, photocopy the passages in question, mark on the page the number and title of the manual, and either FAX or mail the photocopy to Alacron.

Remember to include the name and telephone number of the person we should contact if we have questions.

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
FAX: 603-891-2745

Web site:
http://www.alacron.com/

Electronic Mail:
sales@alacron.com
support@alacron.com