

RW Automation, LLC

**MV Lib**  
**Machine Vision Library**

**USER'S GUIDE**

**Rev 1.1**  
**October 2005**



[www.rwautomation.com](http://www.rwautomation.com)

**Table of Contents**

- 1) Legal Notices ..... 4
  - 1.1) Copyright ..... 4
  - 1.2) License agreement ..... 4
  - 1.3) Limited warranty ..... 4
  - 1.4) Third-party trademarks ..... 4
- 2) Overview ..... 5
  - 2.1) Licensing ..... 5
  - 2.2) Installation ..... 6
  - 2.3) Library naming convention ..... 7
  - 2.4) Error handling ..... 7
- 3) Image I/O ..... 8
  - 3.1) Image types ..... 8
  - 3.2) File formats ..... 9
  - 3.3) mv\_image structure ..... 10
  - 3.4) mv\_image\_file\_header structure ..... 12
  - 3.5) Loading an image from disk ..... 13
  - 3.6) Saving image to disk ..... 14
  - 3.7) Creating a new image ..... 15
  - 3.8) Destroying an image ..... 16
  - 3.9) Copying an image ..... 17
  - 3.10) Duplicating an image ..... 19
  - 3.11) swapping image buffers ..... 21
  - 3.12) Converting an image to a different type ..... 22
- 4) Image graphics ..... 24
  - 4.1) Clearing an image ..... 24
  - 4.2) Drawing primitives ..... 25
  - 4.3) Generating test images ..... 30
  - 4.4) Generating image noise ..... 34
  - 4.5) Dithering images ..... 37
- 5) Image algebra ..... 40
  - 5.1) Add ..... 40
  - 5.2) Subtract ..... 41
  - 5.3) Multiply ..... 41
  - 5.4) Divide ..... 41
  - 5.5) And ..... 41
  - 5.6) Or ..... 41
  - 5.7) Xor ..... 42
  - 5.8) Max ..... 42
  - 5.9) Min ..... 42
- 6) Image transforms ..... 43
  - 6.1) Binary threshold ..... 43
  - 6.2) Negation ..... 44
  - 6.3) Pixel mapping ..... 45
  - 6.4) Contrast stretching ..... 46
  - 6.5) Histogram equalization ..... 47
  - 6.6) Subsampling ..... 51
  - 6.7) Scaling ..... 53
  - 6.8) Scale to fit ..... 54
  - 6.9) Rotation ..... 55
  - 6.10) Mirroring and transposition ..... 57
  - 6.11) Geometric transformation ..... 59
- 7) Filters ..... 61
  - 7.1) Average ..... 62

7.2) Median .....	63
7.3) Sliding Window .....	64
7.4) Sobel.....	66
7.5) Laplacian .....	67
8) Fourier transforms .....	69
8.1) Forward transform.....	69
8.2) Inverse transform .....	70
8.3) Magnitude .....	71
8.4) Phase .....	72
9) Image morphology.....	73
9.1) Max filter.....	73
9.2) Min filter .....	74
10) Normalized correlation search.....	76
10.1) mv_ncs_parameters structure.....	77
10.2) mv_ncs_model structure.....	79
10.3) Train model.....	80
10.4) Search model.....	81
10.5) Destroy model.....	83
11) Image segmentation.....	84
11.1) mv_blob structure.....	84
11.2) Connectivity analysis .....	86
11.3) Multi-level connectivity analysis .....	88
11.4) Watershed analysis.....	89
11.5) Destroying blob arrays .....	91
12) Image projections .....	92
12.1) Computing projections .....	92
12.2) Localizing fiducials.....	94
13) Miscellaneous functions .....	96
13.1) Version number .....	96
13.2) Image statistics .....	97
13.3) Scene angle .....	98
14) Appendix A – Glossary of terms .....	99
15) Appendix B – Error codes .....	101

## **1) Legal Notices**

### **1.1) Copyright**

All material presented in this user's manual is subject to the copyright laws of the United States of America.

Copyright © 2005 RW Automation, LLC - All rights reserved

RW Automation, LLC reserves the right to make changes and improvements to its products and to this document without notification.

### **1.2) License agreement**

The software executables and files packaged with the MV Lib machine vision library are intended for use as-is. Any alteration of the software is strictly forbidden without written permission of RW Automation, LLC. Any bundling, re-packing, or re-distribution of any of the software files included in the original MV Lib distribution file is strictly forbidden without written permission of RW Automation, LLC.

### **1.3) Limited warranty**

RW AUTOMATION, LLC MAKE NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. RW AUTOMATION, LLC WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEROF.

### **1.4) Third-party trademarks**

“Microsoft”, “Microsoft Windows”, and “Visual C++” are registered trademarks of Microsoft Corporation.

## 2) Overview

Welcome to the MV Lib machine vision library. The MV Lib library has the following features:

- Several image types to allow processing of images over wide dynamic ranges.
- Image creation, copy, paste, load from disk, save to disk.
- Image graphics and image generation.
- Image algebra.
- Image transforms, pixel mapping, contrast manipulation, geometric transforms.
- Filters, low pass, high pass, order-statistic, generic sliding window.
- 2-D Fourier transforms.
- Image morphology.
- Normalized correlation search.
- Fiducial localization.
- Image segmentation, connectivity, and watershed analysis.
- Image projections.
- Image statistics and scene angle finder.

The library is written in 'C', and is compiled using the Microsoft Visual C++ 6.0 compiler. The files are compiled as .cpp (C++) files in order to avoid the need to re-cast to "C".

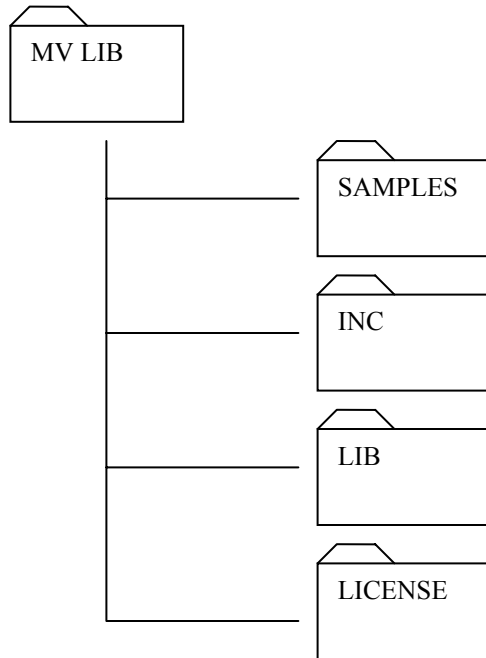
It is assumed that the user has a working knowledge of the 'C' programming language and is familiar with images and how they can be stored and manipulated in memory.

### 2.1) Licensing

Dongle-controlled licensing (using parallel port and USB dongles) is used on a per-copy basis for systems running on a Microsoft Windows operating system.

## 2.2) Installation

The library installation CD has the following directory structure:



To install the library, simply copy it from distribution CD into onto your hard disk at the location of your choice.

The SAMPLES sub-directory holds the MV Tool application, as well as the full source code for building the application under Microsoft Visual C++ 6.0.

The INC sub-directory holds the three header files:

```
mv_functions.h  
mv_definitions.h  
mv_errors.h
```

These three header files must be included in any source files that use the MV Lib functions.

The LIB sub-directory contains the .lib (static library) files for the normal (mv\_lib.lib) and multi-threaded (mv\_lib\_mt.lib) builds of the MV Lib machine vision library.

The LICENSE sub-directory contains the dongle installation files and instructions.

### **2.3) Library naming convention**

All functions within the MV Lib library start with “mv\_”.

All definitions within the MV Lib library start with “MV\_”.

Where possible, function names and parameters were created to be as specific and obvious as possible.

### **2.4) Error handling**

All functions in MV Lib will return an integer status value. The value will allow the user to determine if the function was completed successfully or whether an error occurred.

Appendix B lists all error messages and their probable causes.

### 3) Image I/O

This section describes the use of the image input/output functions.

#### 3.1) Image types

Images used by MV Lib are held in memory. Because of varying processing needs for images, there are multiple types defined for holding images data in memory.

The table below lists the image types defined for MV Lib:

Image type define	Description
MV_IMAGE_TYPE_UNDEFINED	This type is applied to image structures that do not contain any valid image data.
MV_IMAGE_TYPE_BINARY	This type is used for binary images. One byte of buffer space is allocated for each pixel in a binary image, however, only values of 0 and 1 are valid values for a pixel
MV_IMAGE_TYPE_8_BIT	This type is used for 8-bit images. One byte of buffer space is allocated for each pixel in the image and is used to represent the intensity of the pixel from 0 to 255.
MV_IMAGE_TYPE_32_BIT	This type is used for 32-bit images. Four bytes of buffer space is allocated for each pixel in the image and is used to represent the intensity of the pixel from $-2^{31}$ to $+2^{31} - 1$ .
MV_IMAGE_TYPE_DOUBLE	This type is used for double-precision images. Eight bytes of buffer space is allocated for each pixel in the image and is used to represent the intensity of the pixel over the range supported by a double precision floating-point variable.
MV_IMAGE_TYPE_COMPLEX	This type is used for complex images. Sixteen bytes of buffer space is allocated for each pixel in the image. This is used as two eight-byte double precision floating-point variables, one for the real component of the pixel and one for the imaginary component.

MV Lib has many functions that perform operations on images. Many of these functions carry practical limitations on what image types can be used when performing the function. These limitations are listed with each of the functions as they are described in subsequent chapters of this user's guide.

### 3.2) File formats

When reading/writing images from/to disk, the following file formats are supported:

File format define	Description
MV_FILE_FORMAT_UNKNOWN	Reserved.
MV_FILE_FORMAT_GIF	Graphics Interchange Format. Both GIF87 and GIF89a are supported. Color images are converted to 8-bit monochrome images by averaging the red, green, and blue components. GIF images use lossless compression algorithms that tend to work better on graphics images with large areas of uniform intensity or repeating patterns. Real-world images are still compressed, but not as well as graphic images.
MV_FILE_FORMAT_BMP	Windows BitMaP format. Color images are converted to 8-bit monochrome images by averaging red, green, and blue components. No compression is used.
MV_FILE_FORMAT_MV	MV Lib proprietary format. The advantage of this format is that it can be used to store 32-bit, double, and complex image types. No compression is used.
MV_FILE_FORMAT_RAW	This is used for image files in which only the image data will be written. MV Lib can write image files to this format, but cannot read this format.

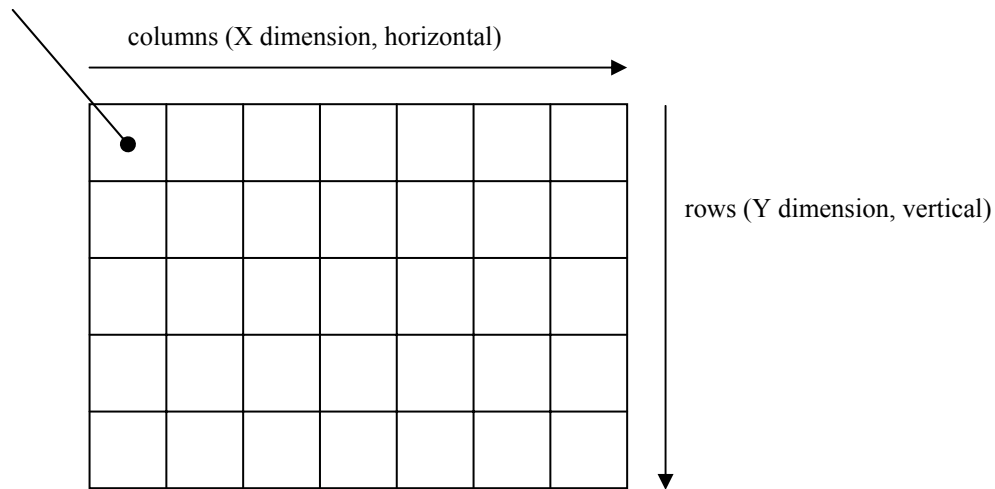
### 3.3) mv\_image structure

The following type definition is used to define the data structure used to hold images in memory. This image structure is used in all functions that require an image to be passed as a parameter.

```
typedef struct mv_image
{
    int columns;
    int rows;
    int type;
    unsigned char *binary_buffer;
    unsigned char *byte_buffer;
    int *integer_buffer;
    double *double_buffer;
    double *real_buffer;
    double *imaginary_buffer;
} mv_image;
```

The columns parameter is used to define the horizontal (X) dimension of an image. The rows parameter is used to define the vertical (Y) dimension of an image. The type parameter is used to define the image type (see section 3.1)

Pixel (x, y) = (0,0)



Example of image coordinates (columns = 7, rows = 5)

Depending on the image type, the appropriate buffer pointers(s) will point to allocated memory of sufficient size (i.e. columns \* rows) of memory elements required to hold all

of the pixel values for the image. Buffer pointers not required for the image type specified will be NULL.

Buffered image data is stored in row-major order. So, for instance, to access point (x, y) in an image of type `MV_IMAGE_TYPE_DOUBLE`, the following code would be used to access the point (without range checking on x and y):

```
double read_point(mv_image *image, int x, int y)
{
    // *image is assume to be initialized to type
    // MV_IMAGE_TYPE_DOUBLE

    return *(image->double_buffer + x + y*image->columns);
}
```

The image buffer is like a two-dimensional array `buffer[y][x]` except, since the buffer is dynamically allocated, it is not practical to have the compiler perform access on 2-D arrays of varying sizes.

### 3.4) `mv_image_file_header` structure

Image files of format `MV_FILE_FORMAT_MV` will start with the following header:

```
typedef struct mv_image_file_header
{
    char label[6];
    short int version;
    int type;
    int columns;
    int rows;
} mv_image_file_header;
```

The `label` must be “MV Lib” (no null terminator)

The `version` is currently 100, and is used to ensure compatibility with future revisions of MV Lib.

The `type` (see section 3.1) determines how many bytes are used to represent the intensity of each pixel. The `columns` and `rows` parameters determine the number of pixels in the image.

After the header is written to the file, bytes of value 0 are written to pad the header to 256 bytes (total header length). Immediately following the header is the image data, written directly from the buffer in memory.

### 3.5) Loading an image from disk

Description:

This function is used to load an image from a disk into memory. The file format of the image is detected automatically.

Function syntax:

```
int mv_disk_to_buffer(mv_image *image,  
                     unsigned char *path_name);
```

Parameters:

**\*image** – Pointer to an uninitialized variable of type `mv_image`.

**\*path\_name** – Pointer to a null-terminated string with path of image file to be opened and read.

Results:

If successful, the variable pointed to by `*image` is initialized to hold the image read from the disk file at specified path name. Otherwise, `*image` is initialized to hold image of type `MV_IMAGE_TYPE_UNDEFINED`.

### 3.6) Saving image to disk

Description:

Writes an image to a disk file.

Function syntax:

```
int mv_buffer_to_disk(mv_image *image,  
                     unsigned char *path_name,  
                     int file_type);
```

Parameters:

**\*image** – Pointer to a variable of type `mv_image`, already initialized with image to be written to disk.

**\*path\_name** – Pointer to a null-terminated string with path of the image file to be opened and written.

**file\_type** – Specifies the file format to be used when writing the image to disk (see section 3.2).

Results:

If successful, the image will be on disk at the specified path. Otherwise, the image will not have been written to disk.

### 3.7) Creating a new image

#### Description:

This is used to create an image in memory. Memory is allocated for the image buffer, but is left un-initialized. It is the user's responsibility to make sure meaningful data is written into the image buffer after it is created using this function.

#### Function syntax:

```
int mv_create_buffer(mv_image *image,  
                    int columns,  
                    int rows,  
                    int type);
```

#### Parameters:

\*image – Pointer to an uninitialized variable of type mv\_image.

columns, rows – Defines size of image.

type – Defines the memory used for each pixel in the image (see section 3.1).

#### Results:

If successful, the variable pointed to by \*image is initialized to hold an image of the size and type specified (the buffer is allocated, but un-initialized). Otherwise, \*image is initialized to hold an image of type MV\_IMAGE\_TYPE\_UNDEFINED.

### 3.8) Destroying an image

Description:

This is used to “destroy” an image. The memory buffer(s) associated with the image are freed and the image is initialized to a safe “undefined” state.

Function syntax:

```
int mv_destroy_buffer(mv_image *image);
```

Parameters:

\*image – Pointer to a variable of type mv\_image, already initialized with image to be destroyed.

Results:

If successful, any allocated memory associated with the image is freed. In any event, the image will be re-initialized to hold an image of type MV\_IMAGE\_TYPE\_UNDEFINED (which requires no allocated memory).

### 3.9) Copying an image

Description:

This function is used to copy a portion of one image into another image. The source and destination images must be of the same type (see section 3.1) in order for this function to work. As well, range checking is used to ensure the copy operation takes place within the allocated image boundaries of the source and destination images.

Function syntax:

```
int mv_copy_buffer(mv_image *src_image,
                  int src_x,
                  int src_y,
                  mv_image *dst_image,
                  int dst_x,
                  int dst_y,
                  int columns,
                  int rows);
```

Parameters:

**\*src\_image** – Pointer to a variable of type `mv_image` that is assumed to be initialized and holding an image to be copied.

**src\_x, src\_y** – The coordinates of the first pixel of the source image to be copied.

**\*dst\_image** – Pointer to a variable of type `mv_image` that is assumed to be initialized (i.e. memory has been allocated).

**dst\_x, dst\_y** – The coordinates of the first pixel of the destination image to be overwritten by the copy operation.

**columns, rows** – The size of the rectangular area over which pixels will be copied from the source image to the destination image. The rectangle is anchored at the respective first pixel coordinates of the source and destination images.

Results:

If successful, the specified region of the source image is copied to the specified region of the destination image.

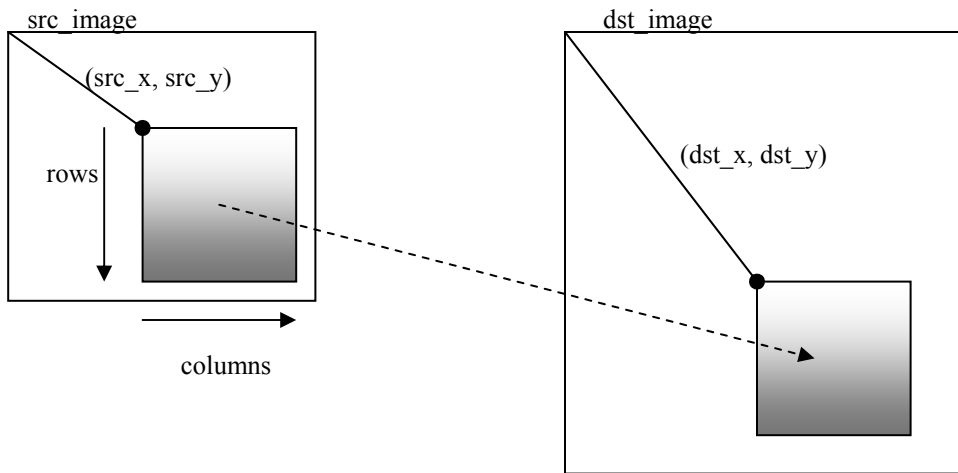


Illustration of parameter usage in `mv_copy_buffer()` function

### 3.10) Duplicating an image

Description:

This function is used to create a new image by copying a portion of an existing image.

Function syntax:

```
int mv_duplicate_buffer(mv_image *src_image,  
                        int src_x,  
                        int src_y,  
                        mv_image *dst_image,  
                        int columns,  
                        int rows);
```

Parameters:

\*src\_image – Pointer to a variable of type mv\_image that is assumed to be initialized and holding an image to be copied.

src\_x, src\_y – The coordinates of the first pixel of the source image to be copied.

\*dst\_image – Pointer to a variable of type mv\_image that is assumed to be uninitialized.

columns, rows – The size of the rectangular area over which pixels will be copied from the source image to the destination image. The rectangle is anchored at the respective first pixel coordinates of the source and destination images.

Results:

If successful, a new image of dimension specified by columns and rows parameters is allocated and initialized using pixel data from the source image. Otherwise, the destination image is initialized to hold an image of type `MV_IMAGE_TYPE_UNDEFINED`.

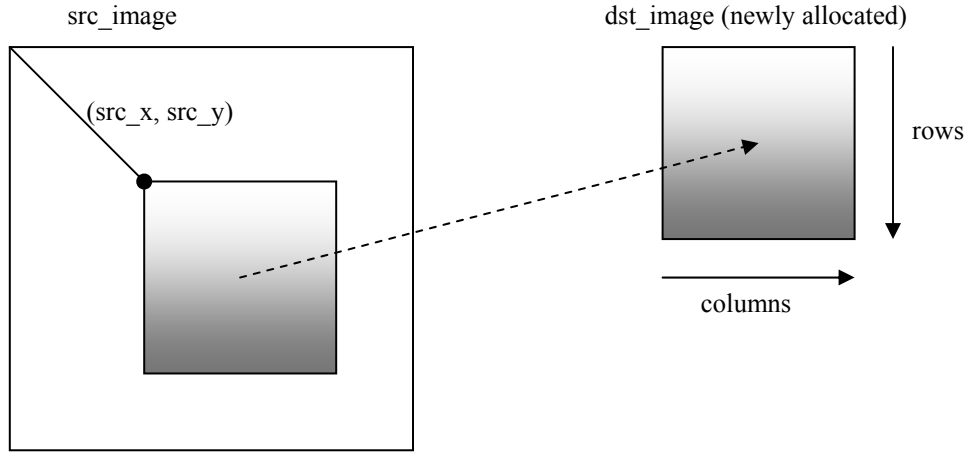


Illustration of parameter usage in `mv_duplicate_buffer` function.

### 3.11) swapping image buffers

Description:

“swaps” two image structures.

Function syntax:

```
int mv_swap_buffers(mv_image *image0, mv_image *image1);
```

Parameters:

\*image0 – Pointer to first variable of type mv\_image.

\*image1 – Pointer to second variable of type mv\_image.

Results:

If successful, this function simply exchanges the two specified mv\_image structures. No additional memory is allocated.

### 3.12) Converting an image to a different type

Description:

This function is used to “convert” an image from one type to another (see section 3.1). Although not required, in most cases, the user will desire to convert an image from a format with lower resolution per pixel to one of higher resolution.

Function syntax:

```
int mv_convert_buffer(mv_image *image, int type);
```

Parameters:

*\*image* – Pointer to variable of type *mv\_image*, assumed to be initialized and holding an image to be converted.

*type* – The image type (see section 3.1) that the image will be converted to.

Results:

If successful, the image will be converted according to the following algorithm:

Old type	New type	Effect
binary	binary	No change
	8-bit	0 is mapped to 0, 1 is mapped to 255
	32-bit	0 is mapped to 0, 1 is mapped to 1
	double	0 is mapped to 0.0, 1 is mapped to 1.0
	complex	0 is mapped to 0.0, 1 is mapped to 1.0 in real component. Imaginary component is set to 0.0
8-bit	binary	0 is mapped to 0, all other values mapped to 1
	8-bit	No change
	32-bit	8-bit value is cast to 32-bit
	double	8-bit value is cast to double
	complex	8-bit value is cast to double in real component. Imaginary component is set to 0.0.
32-bit	binary	Less than or equal to 0 is mapped to 0, all other values mapped to 1.
	8-bit	Maximum and minimum pixel intensities are computed. This is then used in a linear mapping to 8-bit (0-255).
	32-bit	No change
	double	32-bit value is cast to double
	complex	32-bit value is cast to double in real component. imaginary component is set to 0.0.
double	binary	Less than or equal to 0.5 is mapped to 0, all other values mapped to 1.
	8-bit	Maximum and minimum pixel intensities are computed. This is then used in a linear mapping to 8-bit (0-255).
	32-bit	Double is cast to 32-bit.
	double	No change
	complex	Pixel values are transferred to real component. Imaginary component is set to 0.0.
complex	binary	The conversion is performed using the real component only, same as is done for converting a double image. The imaginary component is discarded.
	8-bit	
	32-bit	
	double	
	complex	No change

## 4) Image graphics

MV Lib provides image graphics routines primarily to allow generation of test images for use in developing machine vision applications.

### 4.1) Clearing an image

Description:

This function “clears” an image by setting all of the pixels within the image to the value specified.

Function syntax:

```
int mv_clear_image(mv_image *image, int value);
```

Parameters:

\*image - Pointer to a variable of type mv\_image that is assumed to be initialized.

value – The intensity value that every pixel in the image will be assigned to.

Results:

If successful, all pixels within the image will be set to the value specified. For binary and 8-bit images, the specified value will be logically ANDed with 0x01 and 0xFF respectively when clearing the image. For complex images, the specified value is only written to the real component of each pixel, the imaginary component is set to 0.0.

## 4.2) Drawing primitives

A few simple drawing routines are provided to aid in creating test images.

### 4.2.1) Single pixel

Description:

The following complementary functions allow the user to write and read individual pixels within an image. This means of access image data is relatively inefficient and should only be used for convenience when accessing small groups of pixels. See section 3.3 for details on how to directly access image data.

Function syntax:

```
int mv_set_pixel(mv_image *image,
                int x,
                int y,
                int value);
```

```
int mv_get_pixel(mv_image *image,
                 int x,
                 int y,
                 void *value);
```

Parameters:

*\*image* – Pointer to a variable of type *mv\_image*, assumed to already be initialized.

*x, y* – Coordinates of pixel to be written or read.

*value* – For “set” operation, this holds the value of the pixel to be written.

*\*value* – For “get” operation, this holds a pointer to a variable corresponding to the image type.

Results:

The “set” operation will write the specified value into the pixel at the specified coordinates. The integer value is converted to the image’s type (see section 3.1) using the same algorithm as when clearing an image (see section 4.1). If the specified coordinates are outside of the image boundary, the operation will still return a successful status, but no write operation will have taken place.

If successful, the “get” operation will return the value of the pixel at the specified coordinates. The \*value pointer is cast to the image’s type (see section 3.1) and it is up to the user to ensure that \*value points to a variable of sufficient size to hold the pixel value. Note, in the case of a complex image, only the real component is returned. If the specified coordinates are outside of the image boundary, an error code will be returned (see Appendix B).

## 4.2.2) Lines

Description:

This will draw a line (one pixel wide) between two user-specified points

Function syntax:

```
int mv_draw_line(mv_image *image,  
                int x0,  
                int y0,  
                int x1,  
                int y1,  
                int value);
```

Parameters:

\*image – Pointer to variable of type mv\_image assumed to already be initialized.

x0,y0 – Coordinates of starting point of line.

x1,y1 – Coordinates of ending point of line.

value – Pixel value to be written to image along the specified line.

Results:

A line of single-pixel width will be drawn from point (x0,y0) to point (x1,y1) using the specified pixel value. Any points on the line that are outside of the image boundary will not be written (i.e. they are ignored).

### 4.2.3) Rectangles

Description:

This function draws a rectangle (filled or unfilled) on the image.

Function syntax:

```
int mv_draw_rectangle(mv_image *image,  
                    int x0,  
                    int y0,  
                    int x1,  
                    int y1,  
                    int value,  
                    int fill);
```

Parameters:

\*image – Pointer to variable of type mv\_image assumed to already be initialized.

x0,y0 – Coordinates of one corner of a rectangle.

x1,y1 – Coordinates of second corner of rectangle, opposing point (x0,y0).

value – Pixel value to be written to image to the specified rectangle.

fill – If TRUE, the interior of the rectangle will be “filled” with the specified pixel value. Otherwise, only the outline of the rectangle is drawn on the image using the specified pixel value.

Results:

A rectangle (either filled or an outline of single-pixel width) will be drawn on the image using the specified pixel value. Any points on the rectangle that are outside of the image boundary will not be written (i.e. they are ignored).

#### 4.2.4) Circles

Description:

This function draws a circle (filled or unfilled) on the image.

Function syntax:

```
int mv_draw_circle(mv_image *image,  
                  int x0,  
                  int y0,  
                  int radius,  
                  int value,  
                  int fill);
```

Parameters:

\*image – Pointer to variable of type mv\_image assumed to already be initialized.

x0,y0 – Coordinates of one the center of the circle.

radius – Radius of the circle (in pixels).

value – Pixel value to be written to image for the specified circle.

fill – If TRUE, the interior of the circle will be “filled” with the specified pixel value. Otherwise, only the outline of the circle is drawn on the image using the specified pixel value.

Results:

A circle (either filled or an outline of single-pixel width) will be drawn on the image using the specified pixel value. Any points on the circle that are outside of the image boundary will not be written (i.e. they are ignored).

### 4.3) Generating test images

Description:

This function is used to generate images using a variety mathematical formulae.

Function syntax:

```
int mv_generate_image(mv_image *image,  
                     int function_type,  
                     double x0,  
                     double y0,  
                     double size_x,  
                     double size_y);
```

Parameters:

\*image – Pointer to variable of type mv\_image assumed to be initialized.

function\_type – Defines the mathematical formula to be used when generating the image.

x0, y0, size\_x, size\_y – The meaning of these parameters vary according to the image function specified. The units are always in pixels.

Results:

The following table lists the function types and defines how their respective images are generated:

function type define	description
MV_GENERATE_WHITE_RECTANGLE MV_GENERATE_BLACK_RECTANGLE	Generates a filled rectangle, centered on x0,y0, of width size_x and height size_y.
MV_GENERATE_WHITE_ELLIPSE MV_GENERATE_BLACK_ELLIPSE	Generates a filled ellipse, centered on x0,y0 of with outline defined by:  $\left(\frac{x}{size\_x}\right)^2 + \left(\frac{y}{size\_y}\right)^2 = 1.0$
MV_GENERATE_WHITE_GAUSSIAN_R MV_GENERATE_BLACK_GAUSSIAN_R	Generates a radial Gaussian, centered on x0,y0, of the form:  $value = e^{-2\left(\left(\frac{x}{size\_x}\right)^2 + \left(\frac{y}{size\_y}\right)^2\right)}$
MV_GENERATE_WHITE_SINC_R MV_GENERATE_BLACK_SINC_R	Generates a radial sinc, centered on x0,y0, of the form:  $value = \frac{\sin(r)}{r}$ where:  $r = \sqrt{\left(\frac{x}{size\_x}\right)^2 + \left(\frac{y}{size\_y}\right)^2}$
MV_GENERATE_WHITE_ABSOLUTE_SINC_R MV_GENERATE_BLACK_ABSOLUTE_SINC_R	Generates an absolute radial sinc, centered on x0,y0, of the form:  $value = \frac{ \sin(r) }{r}$ where:  $r = \sqrt{\left(\frac{x}{size\_x}\right)^2 + \left(\frac{y}{size\_y}\right)^2}$
MV_GENERATE_WHITE_SQUARED_SINC_R MV_GENERATE_BLACK_SQUARED_SINC_R	Generates a squared radial sinc, centered on x0,y0, of the form:

	$value = \left( \frac{\sin(r)}{r} \right)^2$ <p>where:</p> $r = \sqrt{\left( \frac{x}{size\_x} \right)^2 + \left( \frac{y}{size\_y} \right)^2}$
<p>MV_GENERATE_WHITE_SIN_R MV_GENERATE_BLACK_SIN_R</p>	<p>Generates a radial sin, centered on x0,y0, of the form:</p> $value = \sin(r)$ <p>where:</p> $r = \sqrt{\left( \frac{x}{size\_x} \right)^2 + \left( \frac{y}{size\_y} \right)^2}$
<p>MV_GENERATE_WHITE_SIN_XY MV_GENERATE_BLACK_SIN_XY</p>	<p>Generates X-Y sin pattern, centered on x0,y0, of the form:</p> $value = \sin\left(\frac{x}{size\_x}\right) \times \sin\left(\frac{y}{size\_y}\right)$
<p>MV_GENERATE_WHITE_CROSS MV_GENERATE_BLACK_CROSS</p>	<p>Generates a crosshair, centered on x0,y0. The vertical and horizontal arms of the cross are size_x and size_y in width respectively. The arms of the cross extend to the edges of the image.</p>

Note: some images are generated based on integrating intensity over a square pixel of unit dimension. Thus, for instance, the edges of a rectangle can show a small gradient rather than a step discontinuity.

For each function type, the contrast (white object on black background, or black object on white background) can be specified.

Internal to MV Lib, the images are generated using double-precision floating-point math and a unity contrast. The resulting values are converted to match the image type of the specified image as follows:

binary – All values > 0.5 map to 1, otherwise, map to 0.

8-bit – All values multiplied by 255 and cast to unsigned 8-bit integer.

32-bit – All values multiplied by 1000 and cast to a signed 32-bit integer

double – no conversion required.

complex – only the real component is generated. The imaginary component is set to 0.0.

In the case of binary and 8-bit images, an intensity offset is added to those functions that can be negative. This will result allow the full intensity range of the generated image to be represented.

#### 4.4) Generating image noise

Description:

This function adds randomly generated noise to an existing image. It is useful in testing the robustness of algorithms under varying levels of noise.

Function syntax:

```
int mv_add_noise(mv_image *image,  
                int noise_type,  
                double parameter1,  
                double parameter2);
```

Parameters:

\*image – Pointer to variable of type mv\_image, assumed to be initialized.

noise\_type – Defines the type of noise to be added to the image.

parameter1, parameter2 – The use of these parameters depend on the type of noise specified

## Results:

The following table lists the noise types and defines how they are generated:

noise type define	description
MV_NOISE_UNIFORM	<p>parameter1 is the mean intensity value of the noise. parameter2 is the standard deviation.</p> <p>For each pixel in the image, random intensity values uniformly distributed over a range of -3 standard deviations to +3 standard deviations, are added to the mean. The resulting noise value is then added to the pixel value.</p>
MV_NOISE_GAUSSIAN	<p>parameter1 is the mean intensity value of the noise. parameter2 is the standard deviation.</p> <p>For each pixel in the image random intensity values, with a Gaussian distribution (approximated by summing 10 uniformly-distributed random values) are added to the mean. The resulting noise value is then added to the pixel value.</p>
MV_NOISE_SALT_AND_PEPPER	<p>parameter1 is the probability (should be in range 0.0 to 1.0) that a pixel will be set to black. parameter2 is the probability (should be in range 0.0 to 1.0) that a pixel will be set to white.</p> <p>For each pixel in the image, a uniformly-distributed random number from 0.0 to 1.0 is generated. If the number is less than parameter1, the pixel is set to 0 (black). If the number is greater than (1.0 – parameter2), the pixel is set to it maximum value (white). Otherwise, the pixel is left as-is.</p>

Note: internally, all noise is generated using double-precision floating-point math. The pixels values for the image are cast as double, then added to the noise value, after which they are converted back to the type for the image as follows:

binary – All values > 0.5 map to 1, otherwise, map to 0.

8-bit – All values are clipped to the range 0 to 255.

32-bit – All values are clipped to the range:  $-(2^{31} + 1)$  to  $2^{31}$ .

double – no conversion required.

complex – only the real component has noise added. The imaginary component is left as-is.

## 4.5) Dithering images

Description:

“Dithering” refers to a means of transforming a grey scale image into a binary image in a manner that appears to preserve the grey scale qualities of the image. In reality, the dithering algorithms make use of the low-pass filtering that is inherent in the human optical system to convert a high frequency binary pattern into the illusion of a continuous grey scale image.

Function syntax:

```
int mv_dither_image(mv_image *src_image,  
                   int columns,  
                   int rows,  
                   mv_image *dst_image,  
                   int dither_type);
```

Parameters:

`*src_image` – Pointer to variable of type `mv_image`. This holds the image to be dithered.

`columns, rows` - The size of the dithered image to be generated.

`dst_image` – Pointer to variable of type `mv_image`. This variable is expected to be uninitialized.

`dither_type` – Specifies dithering algorithm to be used.

**Results:**

If successful, a binary image of the specified size is created using the `dst_image` pointer. The dithered image is then generated using the source image as a reference.

Usually, dithering is only successful when the destination image is substantially larger than the source image. It is the user's responsibility to maintain the correct aspect ratio (if desired) when choosing the size of the dithered image.

Three popular dithering algorithms are available, as shown in the table below:

dither type define	description
<code>MV_DITHER_RANDOM</code>	Each pixel in the destination image is assigned a value (0 or 1) according to a randomly generated number, weighted by the corresponding pixel value of the reference image. Thus, for large areas of uniform pixel value, this method of dithering works well. Smaller high-frequency features tend to get blurred by the random nature of this dithering algorithm.
<code>MV_DITHER_ERROR_DIFFUSION</code>	Error diffusion works by keeping track of error in the difference between the source image and the dithered image. This algorithm does a good job of preserving high-frequency features of an image, but tends to induce visible artifacts into areas of the image composed constant pixel values and low-frequency intensity gradients.
<code>MV_DITHER_HALFTONE</code>	A grid of points is defined for the destination image, and circular dots are placed on the grid with sizes in proportion to the intensity of the source image at the corresponding point. This method is analogous to halftoning used in printing.

Example of image dithering results:

Original image (146 x 70)



Random dither (292 x 140)



Error diffusion (292 x 140)



Halftone (292 x 140)



## 5) Image algebra

Image algebra functions operate on two images, pixel-by-pixel, to produce one image that is a mathematical composite of the two. As such, the two images used must be of the same type and size.

Function syntax:

The following image algebra functions are available:

```
int mv_add_images(mv_image *image0, mv_image *image1);
int mv_subtract_images(mv_image *image0, mv_image *image1);
int mv_multiply_images(mv_image *image0, mv_image *image1);
int mv_divide_images(mv_image *image0, mv_image *image1);
int mv_and_images(mv_image *image0, mv_image *image1);
int mv_or_images(mv_image *image0, mv_image *image1);
int mv_xor_images(mv_image *image0, mv_image *image1);
int mv_max_images(mv_image *image0, mv_image *image1);
int mv_min_images(mv_image *image0, mv_image *image1);
```

Parameters:

**\*image0** – Pointer to variable of type `mv_image`. This holds the first image to be used in the algebraic formula. It is also the destination for the resultant image of the algebraic operation.

**\*image1** – Pointer to variable of type `mv_image`. This holds the second image to be used in the algebraic formula.

Results:

The subsections that follow detail the results for each of the image algebra functions. For complex images separate operations are performed on the real and imaginary components of the image.

### 5.1) Add

The two images are added on a pixel-by-pixel basis and the result is placed back into `image0`. In the case of integer formats, the resultant value is allowed to overflow and “wrap” around if out of the range of the variable. For binary images, the addition operation is equivalent to a logical XOR operation.

### **5.2) Subtract**

Image1 is subtracted from image0 on a pixel-by-pixel basis and the result is placed back into image0. In the case integer formats, the resultant value is allowed to underflow and “wrap” around if out of the range of the variable. For binary images, the subtraction operation is equivalent to an XOR operation.

### **5.3) Multiply**

The two images are multiplied on a pixel-by-pixel basis and the result is placed back into image0. In the case integer formats, the resultant value is allowed to overflow and “wrap” around if out of the range of the variable. For binary images, the multiplication operation is equivalent to a logical AND operation.

### **5.4) Divide**

Image0 is divided by image1 on a pixel-by-pixel basis and the result is placed back into image0. For binary images, the division operation is equivalent to a logical AND operation. No divide-by-zero checking is performed, so if any pixel values in image1 are zero, an indeterminate value will result for that pixel.

### **5.5) And**

This function cannot be performed on double precision or complex images. For other image types a logical AND of the bits in image0 and image1 is performed on a pixel-by-pixel basis and the result is placed back into image0.

### **5.6) Or**

This function cannot be performed on double precision or complex images. For other image types a logical OR of the bits in image0 and image1 is performed on a pixel-by-pixel basis and the result is placed back into image0.

### **5.7) Xor**

This function cannot be performed on double precision or complex images. For other image types a logical XOR of the bits in image0 and image1 is performed on a pixel-by-pixel basis and the result is placed back into image0.

### **5.8) Max**

Image0 and image1 are compared on a pixel-by-pixel basis. The larger of the two pixel values is placed into image0. For binary images, this is the equivalent of performing a logical OR operation.

### **5.9) Min**

Image0 and image1 are compared on a pixel-by-pixel basis. The smaller of the two pixel values is placed into image0. For binary images, this is the equivalent of performing a logical AND operation.

## 6) Image transforms

Various image transforms are supported in MV Lib. Transforms, as the name implies, transform one image into another. No measurements are taken, however, the transformation can make subsequent measurement algorithms function better.

### 6.1) Binary threshold

Unlike the image conversion function of section 3.12, this function allows the user to specify the threshold value used to convert an image to a binary image.

Function syntax:

```
int mv_binarize_image(mv_image *image, double threshold);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be binarized

threshold – The pixel intensity value at or above which will result in a binary value of 1.

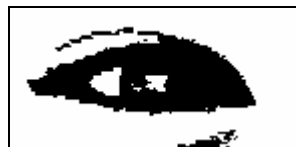
Results:

Each pixel in the specified image has its value checked. If it is greater than or equal to the specified threshold, a binary value of 1 is recorded for that pixel. Otherwise a binary value of 0 is recorded for that pixel. Note, for 8-bit and 32-bit integer images, the threshold value is cast to an integer prior to applying the thresholding algorithm.

At the end of the process, the original image buffer is freed and a pointer to the buffer with the binary data is put into the image structure. The image type is then changed to MV\_IMAGE\_TYPE\_BINARY.



Original



threshold at value of 74

## 6.2) Negation

Negation reverses the contrast of an image. Dark objects become light and light objects become dark.

Function syntax:

```
int mv_negate_image(mv_image *image);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be negated.

Results:

The following transform is applied on a pixel-by-pixel basis to the image according to its type:

binary:  $\text{new\_value} = \text{old\_value} \text{ XOR } 1$

8-bit:  $\text{new\_value} = 255 - \text{old\_value}$

32-bit:  $\text{new\_value} = -\text{old\_value}$

double:  $\text{new\_value} = -\text{old\_value}$

complex:  $\text{new\_real\_value} = -\text{old\_real\_value}$   
 $\text{new\_imaginary\_value} = -\text{old\_imaginary\_value}$



Before negation



After negation

### 6.3) Pixel mapping

Pixel mapping applies only to images of type `MV_IMAGE_TYPE_8_BIT` (see section 3.1).

In this function, each pixel is processed through a user-defined lookup table (LUT) and the pixel value is replaced by the one in the corresponding index of the lookup table.

Function syntax:

```
int mv_pixel_map_image(mv_image *image,  
                      unsigned char pixel_map[256]);
```

Parameters:

`*image` – Pointer to variable of type `mv_image`. This holds the image to be mapped.

`pixel_map[]` – Array of values specified by user to transform image.

Results:

Each pixel in the 8-bit image is changed as follows:

```
new_value = pixel_map[old_value]
```

## 6.4) Contrast stretching

This is a special case of the pixel-mapping function (see section 6.3). This function applies only to images of type MV\_IMAGE\_TYPE\_8\_BIT (see section 3.1).

In this function, the minimum and maximum pixel values for the entire image are determined. A pixel map is then created to “stretch” the pixel values to use the entire dynamic range of the 8-bit image.

Function syntax:

```
int mv_contrast_stretch(mv_image *image);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be transformed.

Results:

Letting MIN and MAX be the minimum and maximum pixel values in the original image, each pixel in the image is then transformed as follows:

$$new\_value = 255 \times \left( \frac{old\_value - MIN}{MAX - MIN} \right)$$

Thus, any pixels with the minimum value will be mapped to 0. Any pixels with the maximum value will be mapped to 255 (the maximum intensity for an 8-bit image). Pixels with values in between the minimum and maximum will have their value scaled in linear proportion to that performed on the pixels with minimum and maximum values.

Note: if MIN equals MAX, there is no contrast in the image, so the image is left as-is.

## 6.5) Histogram equalization

This is a special case of pixel-mapping (see section 6.3). This function applies only to images of type MV\_IMAGE\_TYPE\_8\_BIT (see section 3.1).

Some images are dominated by very high-contrast features that tend to detract from the surrounding areas of lesser contrast. If the histogram (see appendix A) of such an image is viewed, it will be noted that there are one or more dominate modes in the histogram. Histogram equalization uses an algorithm to create a pixel map to “flatten out” the histogram, giving all intensity values roughly equal coverage. In some cases, this transformation can reveal low-contrast features that may not be readily visible because they are mixed with other high-contrast features.

Function syntax:

```
int mv_histogram_equalization(mv_image *image);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be transformed

Results:

A pixel map is created such that the number of pixels in each “bin” of the transformed image’s histogram is approximately the same. The pixel mapping algorithm can be one-to-one or many-to-one, it will never be one-to-many. Thus, since pixels of a given value are never divided into groups of differing values, it is unlikely that a real-world image can ever be mapped to have a truly flat histogram.

Nonetheless, the algorithm will distribute the pixel values as evenly as possible. Once a pixel map has been defined, it is applied to the image as is any other pixel mapping operation (see section 6.3).

Example:

Original image:



Histogram of original image. Vertical axis is the relative frequency of occurrence of each pixel value. The Horizontal axis is the pixel value, with 0 on the left and 255 on the right.

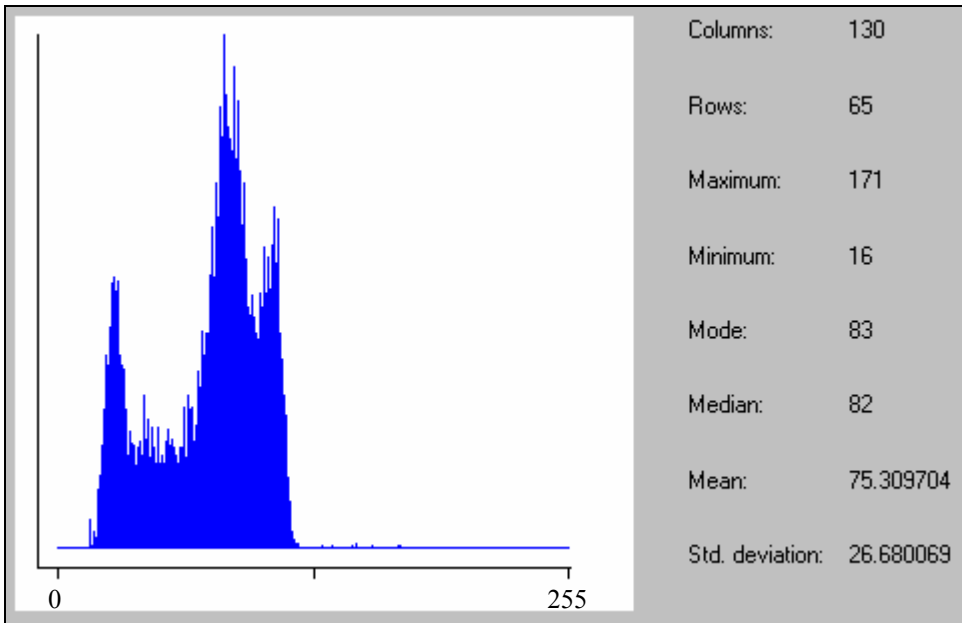


Image after contrast stretch:



Histogram after contrast stretch. The shape of the original histogram is preserved, but it has been “stretched” to fill the full 0-255 range of pixel values.

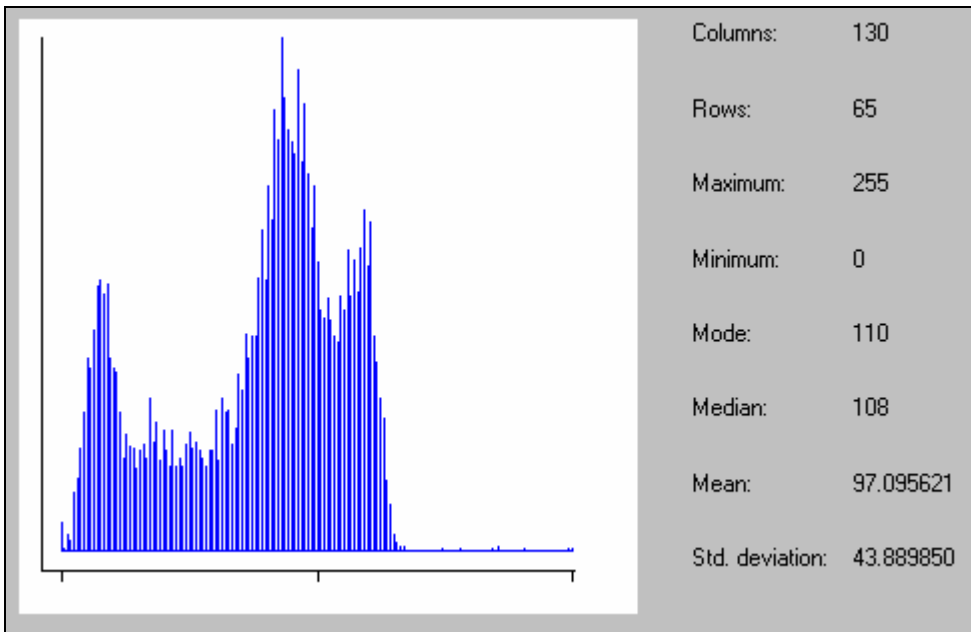
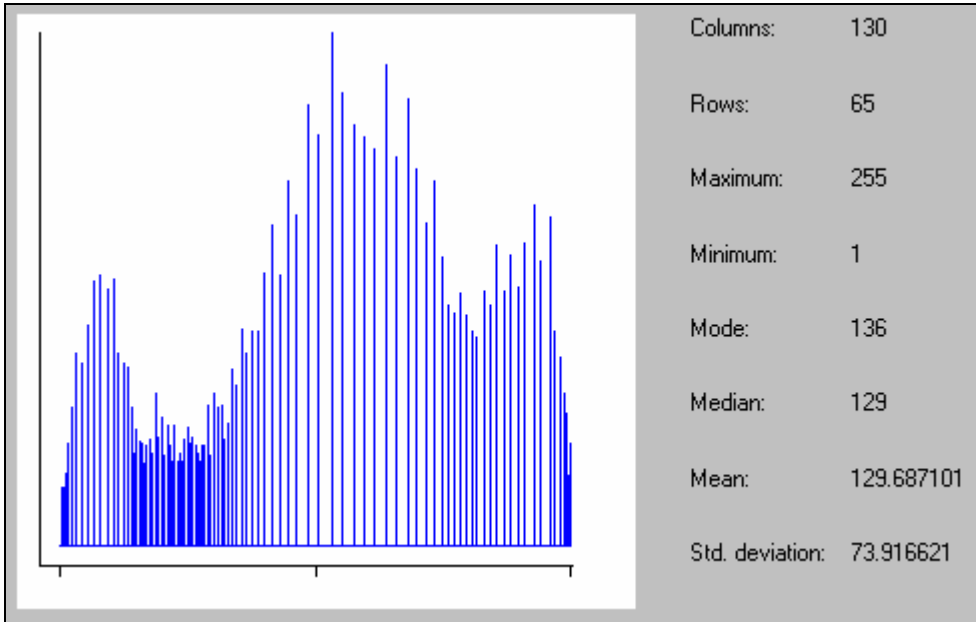


Image after histogram equalization:



Histogram after histogram equalization. Portions of the histogram representing fewer pixels are “compressed” into fewer pixel values. Portions of the histogram representing larger amounts of pixels are expanded to span more of the pixel values.



## 6.6) Subsampling

In this transformation, the image is sub-sampled, independently in X and Y to produce a smaller, lower resolution image. This is a rudimentary form of image scaling (see section 6.7).

Function syntax:

```
int mv_subsample_image(mv_image *image,  
                       int skip_x,  
                       int skip_y);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be transformed

skip\_x – The number of pixels in X to “skip” over (discard) as the image is subsampled.

skip\_y – The number of pixels in Y to “skip” over (discard) as the image is subsampled.

Results:

A new image buffer is allocated with size as follows:

$$new\_columns = 1 + \left( \frac{old\_columns - 1}{skip\_x} \right)$$

$$new\_rows = 1 + \left( \frac{old\_rows - 1}{skip\_y} \right)$$

The mapping from the original image coordinates to the new image coordinates is as follows:

$$new\_image[y][x] = old\_image[y * skip\_y][x * skip\_x]$$

Once the relevant pixels of the original image are copied into the new image buffer (using the above mapping), the original buffer is freed and the new buffer replaces it. The image structure parameters are then adjusted to reflect the image's new size.

Original image:



Subsampled image (skip\_x = 2, skip\_y = 3):



## 6.7) Scaling

Scaling changes the image's size and, in the case where the X and Y scaling factors are different, it will also change the aspect ratio of the image.

Function syntax:

```
int mv_scale_image(mv_image *image,
                  double scale_x,
                  double scale_y,
                  int interpolation_type);

int mv_fast_scale_image(mv_image *image,
                       double scale_x,
                       double scale_y,
                       int interpolation_type);
```

Parameters:

*\*image* – Pointer to variable of type *mv\_image*. This holds the image to be transformed

*scale\_x*, *scale\_y* – The scaling factors defining the transformation.

*interpolation\_type* – Determines how values are interpolated during transformation.

Results:

A geometric transformation matrix of the form:

$$\begin{bmatrix} 1 & 0 \\ \frac{1}{scale\_x} & \\ 0 & \frac{1}{scale\_y} \end{bmatrix}$$

is created and the image is processed using the generic geometric transformation functions (see section 6.11). The original image is destroyed and replaced with the scaled image.

## 6.8) Scale to fit

This function allows the user to conveniently scale an image to an exact size without having to compute the required scale factors.

Function syntax:

```
int mv_scale_image_to_fit(mv_image *src_image,
                        mv_image *dst_image,
                        int interpolation_type);
```

Parameters:

*\*src\_image* – Pointer to variable of type *mv\_image*. This holds the image to be transformed

*\*dst\_image* – Pointer to variable of type *mv\_image*. It is expected to be initialized to the size of the desired (scaled) image.

*interpolation\_type* – Determines how values are interpolated during transformation.

Results:

The scale factors:

$$scale\_x = (\text{double})src\_image \rightarrow \text{columns} / (\text{double})dst\_image \rightarrow \text{columns}$$

$$scale\_y = (\text{double})src\_image \rightarrow \text{rows} / (\text{double})dst\_image \rightarrow \text{rows}$$

are computed. A geometric transformation matrix of the form:

$$\begin{bmatrix} \frac{1}{scale\_x} & 0 \\ 0 & \frac{1}{scale\_y} \end{bmatrix}$$

is created and the image is processed using the generic geometric transformation functions (see section 6.11). The source image is left as-is and the destination image holds the scaled version of the source image.

## 6.9) Rotation

This function allows the user to rotate an image to about a specified origin.

Function syntax:

```
int mv_rotate_image(mv_image *image,
                   double x0,
                   double y0,
                   double angle,
                   int interpolation_type);

int mv_fast_rotate_image(mv_image *image,
                        double x0,
                        double y0,
                        double angle,
                        int interpolation_type);
```

Parameters:

*\*image* – Pointer to variable of type *mv\_image*. This holds the image to be transformed.

*x0, y0* – The center of rotation.

*angle* – The angle of rotation, in radians.

*interpolation\_type* – Determines how values are interpolated during transformation.

Results:

A geometric transformation matrix of the form:

$$\begin{bmatrix} \cos(\textit{angle}) & \sin(\textit{angle}) \\ -\sin(\textit{angle}) & \cos(\textit{angle}) \end{bmatrix}$$

is created. This matrix is then used, along with the specified center of rotation, to compute the translation needed to perform the rotation about the specified center. The image is processed using the generic geometric transformation functions (see section 6.11). The original image is destroyed and replaced with the rotated image.

Example of image rotation.

Original image:



Image rotated +30 degrees (0.5236 radians) about center of image:



## 6.10) Mirroring and transposition

Mirroring and transposition are sometimes useful is changing the coordinate system of an image.

Function syntax:

```
int mv_transpose_image(mv_image *image);
int mv_x_mirror_image(mv_image *image);
int mv_y_mirror_image(mv_image *image);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be transformed.

Results:

A transposition is represented by the following transformation:

$$\text{transposed\_buffer}[y][x] = \text{original\_buffer}[x][y]$$

As can be seen, the image is transformed by exchanging the X and Y coordinate. For images that are not square (i.e. columns and rows are not equal), the transformed image will have its size changed:

$$\text{transposed\_columns} = \text{original\_rows}$$

$$\text{transposed\_rows} = \text{original\_columns}$$

A transposition is equivalent to a 90 degree rotation, followed by a mirroring operation.

Mirroring about the X and Y axes are represented respectively by the following two transformations:

$$\text{x\_axis\_mirrored\_buffer}[y][x] = \text{original\_buffer}[\text{rows} - 1 - y][x]$$

$$\text{y\_axis\_mirrored\_buffer}[y][x] = \text{original\_buffer}[y][\text{columns} - 1 - x]$$

Examples:

Original image:



Y mirrored image:



X mirrored image:



Transposed image:



## 6.11) Geometric transformation

This is a generic geometric transform function. It can be used to skew, rotate, scale, translate, and mirror images in any combination.

In this function, the user must pass a source image to be transformed and a destination image of the same image type into which the transformed image (or portion thereof) is to be placed.

Function syntax:

```
int mv_geometric_transform_image(mv_image *src_image,
                                mv_image *dst_image,
                                double translation_x,
                                double translation_y,
                                double transformation_matrix[2][2],
                                int interpolation_type);
```

```
int mv_fast_geometric_transform_image(mv_image *src_image,
                                       mv_image *dst_image,
                                       double translation_x,
                                       double translation_y,
                                       double transformation_matrix[2][2],
                                       int interpolation_type);
```

Parameters:

**\*src\_image** – Pointer to variable of type `mv_image`. This holds the source image to be transformed.

**\*dst\_image** – Pointer to variable of type `mv_image`. This image must be of the same image type as `src_image`.

**translation\_x, translation\_y** – The translation portion of the geometric transform.

**transformation\_matrix[2][2]** – The 2 x 2 translation matrix.

**interpolation\_type** – Determines how values are interpolated during transformation.

Results:

The image coordinates are transformed as follows:

$$\begin{bmatrix} source\_x \\ source\_y \end{bmatrix} = \begin{bmatrix} translation\_x \\ translation\_y \end{bmatrix} + \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} destination\_x \\ destination\_y \end{bmatrix}$$

Where  $a_{yx}$  are the elements of `transformation_matrix[y][x]`.

For every pixel coordinate in the destination image, the coordinate for the source image is determined using the above equation. If the source image coordinates fall outside of the source image boundary, a value of zero is used for the corresponding destination image pixel value.

Otherwise, there are two types of interpolation. When interpolation type `MV_INTERPOLATION_TYPE_NEAREST_NEIGHBOR` is used, the source coordinate is rounded to the nearest integer value. The pixel value for that coordinate in the source image is then copied to the corresponding coordinate in the destination image.

When interpolation type `MV_INTERPOLATION_TYPE_BILINEAR` is used, the four nearest pixels to the source coordinate are used in a weighted average (once in X and once in Y, hence the term “bilinear”) to produce the pixel value to be placed into the destination image. This interpolation tends to reduce artifacts induced by the sampling process when performing geometric transformations.

Finally, the “fast” version of the geometric transform function has been optimized for binary and 8-bit images, and may only be used with those image types.

## 7) Filters

Various filtering operations are supported in MV Lib. Once an image has been digitized, it is not possible to *create* more information than is already in the image by filtering it. However, filters can be useful at helping to visualize and extract the information that is already in the image.

**Note:** When implementing sliding-window filters, there is always a problem regarding image boundaries. If one desires to keep the filtered image the same size as the original (unfiltered) image, additional algorithms would need to be implemented to interpolate or otherwise approximate the filtered data for the image edges. This becomes more problematic as the filter size is increased.

MV Lib does not attempt to do this. Instead, the filtered image will be smaller than the original image by the dimensions of the filter less 1. For instance, a 10 x 20 image filtered with a 4 x 5 sliding-window filter would result in a filtered image size of 7 x 16.

Also note, that there is an implicit translation of half of the filter size less 1 (in each dimension) between the original and filtered image coordinates.

## 7.1) Average

The averaging filter function is a low-pass filter. It consists of a rectangular filter window over which each filter coefficient is weighted equally. The sum of all filter coefficients is unity. This is a special case of the generic sliding window filter (see section 7.3).

Function syntax:

```
int mv_averaging_filter(mv_image *image,  
                       int filter_columns,  
                       int filter_rows);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be filtered.

filter\_columns, filter\_rows – The size of the averaging filter.

Results:

The original image is filtered using the specified averaging filter size and the filtered image is returned in the original image structure. The image type remains the same, and hence the filtered values for images of lower resolution image types will be rounded as needed to accommodate the image intensity resolution.

Original image:



After 4x4 averaging filter:



## 7.2) Median

The median filter will rank (order) all pixel values within the filter window and will select the median pixel value. Although still a sliding window filter, this is a non-linear operation that differs from most sum-of-product filters.

The median filter can be helpful at removing certain types of noise while preserving edges and other high-contrast features in the image.

Function syntax:

```
int mv_median_filter(mv_image *image,  
                    int filter_columns,  
                    int filter_rows);
```

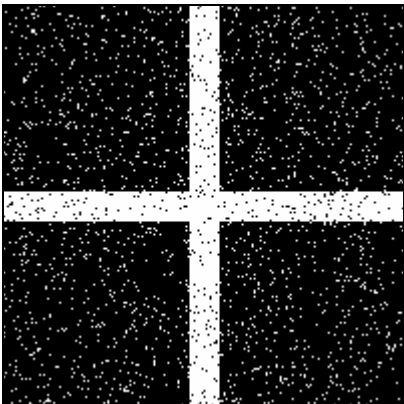
Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be filtered.

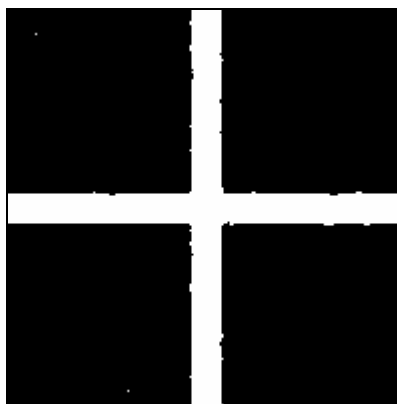
Results:

The original image is filtered using the specified median filter size and the filtered image is returned in the original image structure. Since the filtered pixel values are copied directly from a single (median) pixel value in the original image, there is no intensity resolution issue with this filter.

Original image:



After 3x3 median filter:



### 7.3) Sliding Window

This function allows users to define their own sliding-window filters. The filter itself is just represented by another image that is then cross-correlated with the image to be filtered.

This function will not work with binary images.

Function syntax:

```
int mv_sliding_window_filter(mv_image *image,
                             mv_image *filter,
                             int operation);
```

Parameters:

*\*image* – Pointer to variable of type *mv\_image*. This holds the (source) image to be filtered.

*\*filter* – Pointer to variable of type *mv\_image*. This holds the filter to be applied to the image to be filtered.

*operation* – The filtering operation to be performed.

Results:

When the operation is set to `MV_FILTER_OPERATION_SUM_OF_PRODUCTS`, the filter is stepped (slid) over the image to be filtered. At each pixel location, the source image pixel values are multiplied by the filter coefficients. The sum of all of these products is then the filtered value that is stored into the destination image.

When the operation is set to `MV_FILTER_OPERATION_GRADIENT_MAGNITUDE`, the filter is assumed to be suitable for computing a derivative in the X direction. The transpose of the filter is then suitable for computing a derivative in the Y direction. In the case of a non-square filter, the overall filter size is treated as that of a square filter with dimension equal to the larger of the two dimensions of the filter. The X derivative and Y derivative filters are used independently as sum-of-product filters to compute two images (one filtered with the X derivative and one filtered with the Y derivative). The “gradient magnitude” is then computed as:

$$magnitude = \sqrt{dx^2 + dy^2}$$

where  $dx$  is the value of the pixel filtered using the X derivative filter and  $dy$  is the value of the pixel filtered using the Y derivative filter. The magnitude is then entered into the destination image.

When the operation is set to `MV_FILTER_OPERATION_GRADIENT_ORIENTATION`, the same derivative process is used to filter the image that was used in the gradient magnitude operation. However, rather than computing the gradient, the orientation is computed as:

$$\theta = \tan^{-1}\left(\frac{dy}{dx}\right)$$

The orientation,  $\theta$  (in radians), is then entered into the destination image. Note that for 8-bit and 32-bit image types, the resulting truncation is relatively harsh. It is likely better to perform the gradient orientation using a double-precision image and then convert it back to one of the integer formats.

In all cases, the original source image is destroyed and is replaced with the filtered destination image.

## 7.4) Sobel

The Sobel X derivative filter is defined as:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The Sobel Y derivative is the transpose of the X derivative.

Function syntax:

```
int mv_sobel_filter(mv_image *image, int operation);
int mv_fast_sobel_edge_filter(mv_image *image);
```

Parameters:

*\*image* – Pointer to variable of type *mv\_image*. This holds the image to be filtered.

*operation* – The filtering operation to be performed.

Results:

While it is allowed to use the sum-of-products operation, it will produce an image filtered only with the X derivative Sobel filter and may not be very meaningful.

When using the gradient magnitude or gradient orientation operations, the operation is performed using double-precision math and the results are scaled back to the full resolution of the image type.

The “fast” Sobel edge filter function performs only the gradient magnitude operation and only works on 8-bit images. It is optimized for 8-bit images and uses the following approximation to the magnitude calculation:

$$magnitude = \left( \frac{|dx| + |dy|}{2} \right)$$

The resulting magnitude is clipped to the range 0 to 255.

## 7.5) Laplacian

The Laplacian filter is a high-pass filter defined by:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Function syntax:

```
int mv_fast_laplacian_filter(mv_image *image);
```

Parameters:

*\*image* – Pointer to variable of type *mv\_image*. This holds the image to be filtered.

Results:

A sum-of-product operation is performed using the Laplacian 3 x 3 filter. This function is optimized for 8-bit images and will only work with 8-bit images. The pixel values resulting from the filter are offset by +128 and the result is clipped to the range 0 to 255. The resulting filtered image is returned in the same structure used to pass the source image. The source image is destroyed.

Examples:

Original image:



After Laplacian filter:



After Sobel gradient magnitude filter:



After Sobel gradient orientation filter:



## 8) Fourier transforms

Two-dimensional Fourier transforms of images can be of help in understanding the frequency components of an image source.

### 8.1) Forward transform

The discrete form of the forward Fourier transform (DFT):

$$F(n) = \sum_{k=0}^{N-1} f(k) \left( e^{\frac{j2\pi}{N}} \right)^{-kn}$$

is used successively on the rows and columns of a complex image to compute the forward Fourier transform of the image. When the number of columns and/or rows of an image is a power of two, a fast Fourier transform (FFT) is used in the place of the DFT to improve processing speed.

Function syntax:

```
int mv_forward_fourier_transform(mv_image *image);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be transformed.

Results:

The image type used must be complex (see section 3.1). The discrete Fourier transform is applied to the image and the result is returned in the same image structure.

## 8.2) Inverse transform

The discrete form of the inverse Fourier transform:

$$f(k) = \frac{1}{N} \sum_{n=0}^{N-1} F(n) \left( e^{\frac{j2\pi}{N}kn} \right)$$

is used successively on the rows and columns of a complex image to compute the inverse Fourier transform of the image. When the number of columns and/or rows of an image is a power of two, a fast Fourier transform (FFT) is used in the place of the DFT to improve processing speed.

Function syntax:

```
int mv_inverse_fourier_transform(mv_image *image);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be transformed.

Results:

The image type used must be complex (see section 3.1). The inverse form of the discrete Fourier transform is applied to the image and the result is returned in the same image structure.

### 8.3) Magnitude

This function accepts a complex image (presumably the Fourier transform of an image) and converts it to a double precision image (see section 3.1) of the magnitude of the Fourier transform. The magnitude of each pixel is defined by:

$$magnitude = \sqrt{real^2 + imaginary^2}$$

Function syntax:

```
int mv_convert_to_magnitude(mv_image *image);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be converted.

Results:

A double precision image of the magnitude of the complex image is returned in the image structure passed to this function.

## 8.4) Phase

This function accepts a complex image (presumably the Fourier transform of an image) and converts it to a double precision image (see section 3.1) of the phase of the Fourier transform. The phase of each pixel is defined by:

$$phase = \tan^{-1}\left(\frac{imaginary}{real}\right)$$

The phase is in units of radians and accounts for the full  $2\pi$  range of phase values.

Function syntax:

```
int mv_convert_to_phase(mv_image *image);
```

Parameters:

*\*image* – Pointer to variable of type *mv\_image*. This holds the image to be converted.

Results:

A double precision image of the phase of the complex image is returned in the image structure passed to this function.

## 9) Image morphology

Morphological filters are used to widen or thin lines, usually as a precursor to boundary following and binary segmentation algorithms.

### 9.1) Max filter

A sliding window filter (see section 7) is applied to the image. Within the window, the pixels are ordered by intensity and the pixel value of maximum intensity is chosen as the new value for the pixel.

Function syntax:

```
int mv_max_filter(mv_image *image,
                 int filter_columns,
                 int filter_rows);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be filtered.

filter\_columns, filter\_rows – The size of the max filter.

Results:

The filtered image is returned in the same image structure that was passed to this function. As with other sliding window filters, the resulting image size is reduced by the size of the filter, less 1, when compared to the original.

For complex images, the max filter is applied separately to both the real and imaginary components.

## 9.2) Min filter

A sliding window filter (see section 7) is applied to the image. Within the window, the pixels are ordered by intensity and the pixel value of minimum intensity is chosen as the new value for the pixel.

Function syntax:

```
int mv_min_filter(mv_image *image,  
                 int filter_columns,  
                 int filter_rows);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be filtered.

filter\_columns, filter\_rows – The size of the min filter.

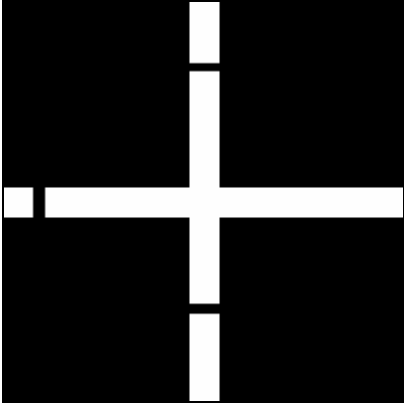
Results:

The filtered image is returned in the same image structure that was passed to this function. As with other sliding window filters, the resulting image size is reduced by the size of the filter, less 1, when compared to the original.

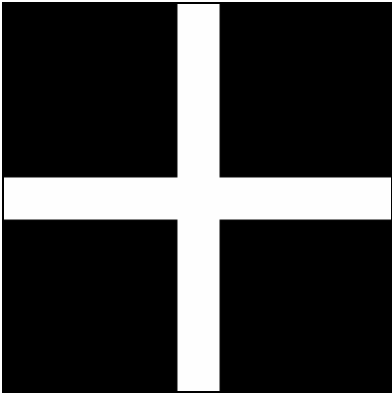
For complex images, the min filter is applied separately to both the real and imaginary components.

Example:

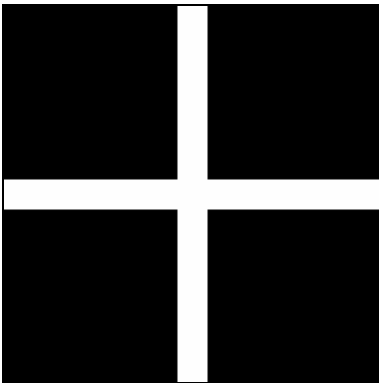
Original “broken cross” image:



After application of 7x7 max filter:



7x7 min filter applied following max filter. Cross width is now back to original, but broken arms of cross have been filled:



## 10) Normalized correlation search

Normal Correlation Search (NCS) is a tool used to allow users to “train” a vision system to recognize a specific pattern (or “model”) in an image, usually by presenting the system with a known-good image of the model.

Once trained, the user can search for similar instances of the same model within an image. The NCS search function is capable of determining the position of the model to sub-pixel accuracy.

Note: all NCS functions operate only with binary and 8-bit images.

Normalized correlation between two images (a model and an image being searched) is defined by:

$$ncs\_map[j][i] = \frac{\left( \frac{\sum_{l=0}^{N-1} \sum_{k=0}^{M-1} (P[l][k] \times I[j+l][i+k])}{M \times N} - (\bar{P} \times \overline{I[j][i]}) \right)^2}{\left( \frac{\sum_{l=0}^{N-1} \sum_{k=0}^{M-1} (P[l][k])^2}{M \times N} - (\bar{P})^2 \right) \times \left( \frac{\sum_{l=0}^{N-1} \sum_{k=0}^{M-1} (I[j+l][i+k])^2}{M \times N} - (\overline{I[j][i]})^2 \right)}$$

Where:

$P[l][k]$  is the model image (of size  $M \times N$ ).

$I[j][i]$  is the image being searched.

$\bar{P}$  is the average model image value (taken over the entire MxN area of the model image).

$\overline{I[j][i]}$  is the average value of the search image in a MxN area starting at pixel  $[j][i]$

$ncs\_map[j][i]$  is the resulting normalized correlation “map”. Values that are close to unity indicate that the model and image are highly correlated at that point.

### 10.1) `mv_ncs_parameters` structure

Correlation of two images is a computationally intensive process. In order to reduce the time to complete the NCS map, the image can first be correlated to the model at lower spatial resolutions. Points in the image with high probabilities of matching the model are then processed at successively higher spatial resolutions. The final determination of the model position is always done at the best resolution.

Several user-selectable parameters affect the performance of the normalized correlation search algorithm. Most of them require the user to consider the tradeoffs between speed and ability to find the target pattern. The structure below encapsulates the search parameters that can be set by the user.

```
typedef struct mv_ncs_parameters
{
    int rank;
    int image_subsample_factor;
    int map_subsample_factor;
    int potential_matches;
} mv_ncs_parameters;
```

The variables in the parameters structure are used as follows:

`rank` - this parameter allows the user to search for lesser ranked targets. Rank 0 is used for the best target, rank 1 for 2nd best, and so on. Normally the user is only interested in the best (rank 0) match, but occasionally lesser ranked matches are of interest. So, if the user expects an image to have multiple instances of the same model and there is a desire to find them all, the search function would be run multiple times with varying rank values to separate the models.

`image_subsample_factor` - specifies how much subsampling is to be used on the image (both search image and model must be subsampled equally in order for a meaningful computation). A factor of 0 is for no subsampling, a factor of 1 samples every other pixel, a factor of 2 samples every 3rd pixel, and a factor of 3 samples every 4th pixel. This factor must be in the range 0 to 3.

`map_subsample_factor` - in addition to subsampling the images used to create the NCS "map". The NCS map itself can be subsampled as it is computed at various resolutions. The effect is similar to subsampling the image, but not identical. This factor must be in the range 0 to 3.

`potential_matches` - When an NCS map is generated by subsampling either (or both) the image and the map, then the best local maxima generated with the subsampling process is not necessarily the one that would've been chosen under a full resolution search. Thus,

increasing the `potential_matches` value will cause more sites (potential matches) from the low resolution map to be evaluated at full resolution. This, of course, will cost more CPU time but can result in more reliable results if similar targets appear in the image with the actual target.

## 10.2) `mv_ncs_model` structure

A normalized correlation search model is the result of the training process for the normalized correlation search. The image used to train the model is subsampled at various resolutions and some sums are pre-computed. This will save time during the actual search since, in most cases, the model is trained once and used many times to search.

```
typedef struct mv_ncs_model
{
    mv_image image[4];
    unsigned int sum[4];
    unsigned int sum2[4];
} mv_ncs_model;
```

From the user's perspective, the only relevant portion of this model structure is the image stored in `image[0]` of the model structure. This image is a full resolution copy of the original training image used to create the model. By storing this image on a disk (see section 3.6), it can later be loaded and used to train the model again.

### 10.3) Train model

This is the function used to “train” a NCS model. An image of size no greater than 256 x 256 pixels is passed to this function, and is expected to hold an image (model) that can be found in subsequent images.

How to select an image for training a model:

uniqueness – The model image should contain a unique pattern or portion of a scene that will not occur in other locations, even in part.

vertical and horizontal components – In order to localize the model in two dimensions, the model image should have both horizontal and vertical features. A set of vertical bars would be good for localizing a target in X, but not in Y. Orthogonal angled features and curved features are usually acceptable as well.

features of varying widths – The features in an image should be of varying width. If the model image contains too many very fine (1 or 2 pixel) thick lines, it will be difficult to subsample the image to obtain efficient computational speed.

varying pitch between edges – A very regular pattern of edges will produce a NCS map with several maxima in close proximity to the true match. By varying the pitch between edges, this problem can be greatly mitigated.

Function syntax:

```
int mv_ncs_train(mv_image *image, mv_ncs_model *model);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be used to “train” the model.

\*model – Pointer to variable of type mv\_ncs\_model. This variable is expected to be uninitialized.

Results:

The model structure is returned fully initialized with the model derived from the image passed to the function.

## 10.4) Search model

Once a model has been trained, the user may search for it in any image by using this function.

Note the image used must be at least 4 pixels in size (both X and Y) greater than the size of the image used in training the model.

Factors affecting search:

uniqueness – If the model is not unique within the image being searched, then the probability of a mismatch increases.

rotation – Except in the case where a the dominant feature(s) of a model image are rotationally symmetric (for instance, a disk, or concentric rings), the performance of the normalized correlation algorithm will tend to degrade rapidly as the angle between the image and the model increases.

focus/scale – Normalized correlation depends on the scale of the model to remain constant from the model image to the image being searched. The performance of the normalized correlation algorithm will tend to degrade as the relative scale between the image and the model increases. As well, if the dominant features of the model do not contain symmetry in both the X and Y axes, the measure location of the model can shift as the scale is changed.

noise – Normalized correlation is reasonably tolerant of noise and will degrade steadily as noise is increased.

contrast reversal – Normalized correlation search will not succeed when a contrast reversal occurs.

A common reaction by users to failures of NCS is to exclaim “The target is *right there!* Why doesn’t the NCS algorithm *see* it?!”. What the novice user fails to understand is that the human visual system is very adept at compensating for scale, rotation, asymmetries, and noise. The NCS algorithm is purely mathematical, it cannot “understand” the underlying structure of the image.

Function syntax:

```
int mv_ncs_search(mv_image *image,
                 mv_ncs_model *model,
                 mv_ncs_parameters *parameters,
                 double *x,
                 double *y,
                 double *score);
```

Parameters:

\*image – Pointer to variable of type mv\_image. This holds the image to be searched.

\*model – Pointer to variable of type mv\_ncs\_model. This holds the model to be used in searching the image.

\*parameters – Pointer to variable of type mv\_ncs\_parameters. This holds the parameters to be used in conducting the search.

\*x, \*y – Holds the location of the model found.

\*score – Holds the normalized correlation value (the “score”).

Results:

If not successful due to an error, or if the model is not found at all, the score will be 0.0 and the X and Y locations will also be set to 0.0.

Otherwise, the X and Y location (in pixels, to sub-pixel accuracy) are returned. Note, the location represents the offset between pixel (0,0) of the search image and pixel (0,0) of the model image.

The normalized correlation score is a number from 0.0 to 1.0. A perfect score of 1.0 is only achieved if the model and search image (where the model is found) are identical. Usually, for real-world images, scores of 0.75 and above are considered good. Scores below 0.5 usually indicate a poor match. It is up to the users to determine the pass/fail threshold on the score that should be set to best suit their applications.

### **10.5) Destroy model**

Use this function to destroy a NCS model once it is no longer needed.

Function syntax:

```
int mv_ncs_destroy_model(mv_ncs_model *model);
```

Parameters:

\*model – Pointer to variable of type mv\_ncs\_model. This holds the model to be destroyed.

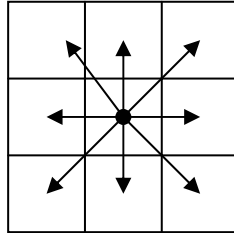
Results:

All memory associated with the model is freed.

## 11) Image segmentation

The image segmentation functions of MV Lib divide an image into disjoint, non-overlapping groups of connected pixels, referred to as “blobs”.

When defining connectivity, MV Lib uses the 8-connected model in which a pixel is considered to be adjacent to all 8 surrounding pixels (up/down, left/right, diagonally).



### 11.1) mv\_blob structure

The mv\_blob structure is used to define the pixels included in a blob. As well, additional useful measures of the blob are included in the structure.

```
typedef struct mv_blob
{
    int area;
    unsigned short int *pixels_x;
    unsigned short int *pixels_y;
    int min_x;
    int max_x;
    int min_y;
    int max_y;
    double center_of_mass_x;
    double center_of_mass_y;
    int fill_level;
    int max_depth;
    double average_depth;
} mv_blob;
```

The parameters of the blob structure are as follows:

area – This is the area of the blob, in pixels.

`*pixels_x`, `*pixels_y` – These point to memory arrays allocated to store the coordinates of all pixels in the blob. The area parameter defines the size of the arrays. The pixels are not guaranteed to be arranged in any particular order. The following code fragment illustrates how to use the coordinate arrays to “paint” a blob onto an image:

```
void paint_blob(mv_image *image, mv_blob *blob, int color)
{
    for (int i = 0; i < blob->area; ++i)
    {
        mv_set_pixel(image,
                     (int) (* (blob->pixels_x + i)),
                     (int) (* (blob->pixels_y + i)),
                     color);
    }
}
```

`min_x`, `max_x`, `min_y`, `max_y` – These parameters define the bounding rectangle of the blob. As each blob pixel is entered into the coordinate arrays, the minimum and maximum coordinate values are recorded independently in X and Y.

`center_of_mass_x`, `center_of_mass_y` – This is the geometric centroid of the blob. Taken by averaging the coordinates of all blob pixels independently in X and Y.

`fill_level` – Used in watershed analysis to indicate the pixel value that the blob has been “filled” to.

`max_depth` – Used in watershed analysis to indicate the maximum “depth” of all pixels in the blob.

`average_depth` – Used in watershed analysis to indicate the average “depth” of the pixels in the blob.

## 11.2) Connectivity analysis

This function only works on binary images. In this case the blobs are assumed to be white (pixel value of 1) on a black background (pixel value of 0). If the user desires to find black blobs on a white background, the image must be negated first (see section 6.2).

In connectivity analysis, each white pixel found is grouped with adjacent white pixels (using 8-connected definition of adjacent) to form blobs. The user can limit the size and number of blobs selected.

Function syntax:

```
int mv_connectivity_analysis(mv_image *image,
                             int max_blobs,
                             int min_area,
                             int max_area,
                             mv_blob **blob_array,
                             int *num_blobs_found);
```

Parameters:

**\*image** – Pointer to variable of type `mv_image`. This holds the image to have a connectivity analysis run on it.

**max\_blobs** – Specifies the maximum number of blobs to be found. If more blobs are found than this number, the smallest ones (least area) are discarded.

**min\_area, max\_area** – Specifies the minimum and maximum area permitted for a blob. Blobs that are outside these limits are ignored.

**\*\*blob\_array** – A pointer to a variable of type `*mv_blob`. An array of `mv_blob` structures will be allocated and initialized as the blobs are found. This array will vary in size to accommodate the blobs found.

**\*num\_blobs\_found** – A pointer to a variable of type `int`. It will be returned holding the number of blobs found and will match the number of blobs allocated for the `blob_array` variable.

Results:

An array of blobs is returned. The blobs returned will adhere to the area limits set by the user and the maximum number of blobs specified by the user. The blobs will be ordered with the blobs of largest area being the first in the array.

Referring to the `paint_blob()` function listed in section 11.1, the following code example below illustrates how to paint all the blobs returned after a connectivity analysis.

```
void paint_all_blobs(mv_image *image,
                    mv_blob *blob_array,
                    int num_blobs_found,
                    int color)
{
    for (int i = 0; i < num_blobs_found; ++i)
    {
        paint_blob(image, (blob_array + i), color);
    }
}
```

### 11.3) Multi-level connectivity analysis

This function is very similar to connectivity analysis. It will operate only on 8-bit images. In this function, a pixel value of 0 is still considered to be the background. Blobs are defined as 8-connected areas of the same pixel value. Thus, unlike the binary case of connectivity analysis, the blobs in multi-level connectivity analysis can be adjacent to each other.

Function syntax:

```
int mv_multilevel_connectivity_analysis(mv_image *image,
                                       int max_blobs,
                                       int min_area,
                                       int max_area,
                                       mv_blob **blob_array,
                                       int *num_blobs_found);
```

Parameters:

**\*image** – Pointer to variable of type `mv_image`. This holds the image to have a multi-level connectivity analysis run on it.

**max\_blobs** – Specifies the maximum number of blobs to be found. If more blobs are found than this number, the smallest one are discarded.

**min\_area, max\_area** – Specifies the minimum and maximum area permitted for a blob. Blobs that are outside these limits are ignored.

**\*\*blob\_array** – A pointer to a variable of type `*mv_blob`. An array of `mv_blob` structures will be allocated and initialized as the blobs are found.

**\*num\_blobs\_found** – A pointer to a variable of type `int`. It will be returned holding the number of blobs found and will match the number of blobs allocated for the `blob_array` variable.

Results:

An array of blobs is returned. The blobs returned will adhere to the area limits set by the user and the maximum number of blobs specified by the user. The blobs will be ordered with the blobs of largest area being the first in the array.

## 11.4) Watershed analysis

The watershed analysis treats the image as a 3-D topology, with pixel intensity value being the Z-axis. If one can imagine pouring water onto such a topology, it would run down from areas of higher pixel values and collect into pools at the areas of lower pixel values.

In a watershed analysis, all local minima in the image (even those comprised of multiple 8-connected pixels of the same value) are the seeds of a pool of water. Each pool is “filled” with water until it is just about to spill over into another pool (or off the edge of the image).

Once so filled, the surface areas of the pools are then recorded in a blob array, just as with a connectivity analysis. The “depth” of the pool at each pixel location is the level (pixel value) of the pool when filled, minus the original pixel value. Thus, deep pools represent features of higher contrast than shallow pools.

In the watershed analysis function, the pools are actually filled in the original image. Thus, successive applications of the watershed analysis may be run on the image to combine the initial pools into larger ones.

Water that runs to the edge of the image will fall off the edge, thus the process of successively filling the pools is ultimately limited by the image edges.

Function syntax:

```
int mv_watershed_analysis(mv_image *image,
                        int max_blobs,
                        int min_area,
                        int max_area,
                        mv_blob **blob_array,
                        int *num_blobs_found);
```

Parameters:

**\*image** – Pointer to variable of type `mv_image`. This holds the image on which the analysis is to be performed.

**max\_blobs** – Specifies the maximum number of blobs to be found. If more blobs are found than this number, the smallest one are discarded.

**min\_area, max\_area** – Specifies the minimum and maximum area permitted for a blob. Blobs that are outside these limits are ignored.

**\*\*blob\_array** – A pointer to a variable of type **\*mv\_blob**. An array of **mv\_blob** structures will be allocated and initialized as the blobs are found.

**\*num\_blobs\_found** – A pointer to a variable of type **int**. It will be returned holding the number of blobs found and will match the number of blobs allocated for the **blob\_array** variable.

#### Results:

An array of blobs is returned. The blobs returned will adhere to the area limits set by the user and the maximum number of blobs specified by the user. The blobs will be ordered with the blobs of largest area being the first in the array.

The original image is altered to reflect the filling function of the watershed analysis.

### 11.5) Destroying blob arrays

This function destroys a blob array by freeing all memory allocated for the array (including the array itself).

Function syntax:

```
int mv_destroy_blob_array(mv_blob *blob_array,  
                          int num_blobs);
```

Parameters:

**\*blob\_array** – Pointer of type **\*mv\_blob** to an array of blobs allocated using one of the connectivity analysis or watershed analysis functions.

**num\_blobs** – The number of blobs in the array.

Results:

All memory for each of the blobs in the array is freed. Then the memory for the blob array is freed.

## 12) Image projections

“Projecting” an image is a means by which a 2-dimensional image is reduced to a 1-dimensional “projection” of the image by summation of pixel values onto a specific axis. For instance, projecting an image onto the X axis requires summing the pixel values for each column of the image.

### 12.1) Computing projections

These two functions are used to compute the projection of an image. The first function listed below allows the user to project the image onto any arbitrary axis. The summation process weighs the values in the 1-D projection according to the number of pixels used in the summation. The second function listed below projects the image onto a specific axis (X or Y) and is optimized for these special cases.

Function syntax:

```
int mv_projection(mv_image *image,
                 mv_image *projection,
                 double angle);

int mv_projection_onto_axis(mv_image *image,
                           mv_image *projection,
                           int axis);
```

Parameters:

**\*image** – Pointer to variable of type `mv_image`. This holds the image to be projected.

**\*projection** – Pointer to a variable of type `mv_image`. This variable should be uninitialized.

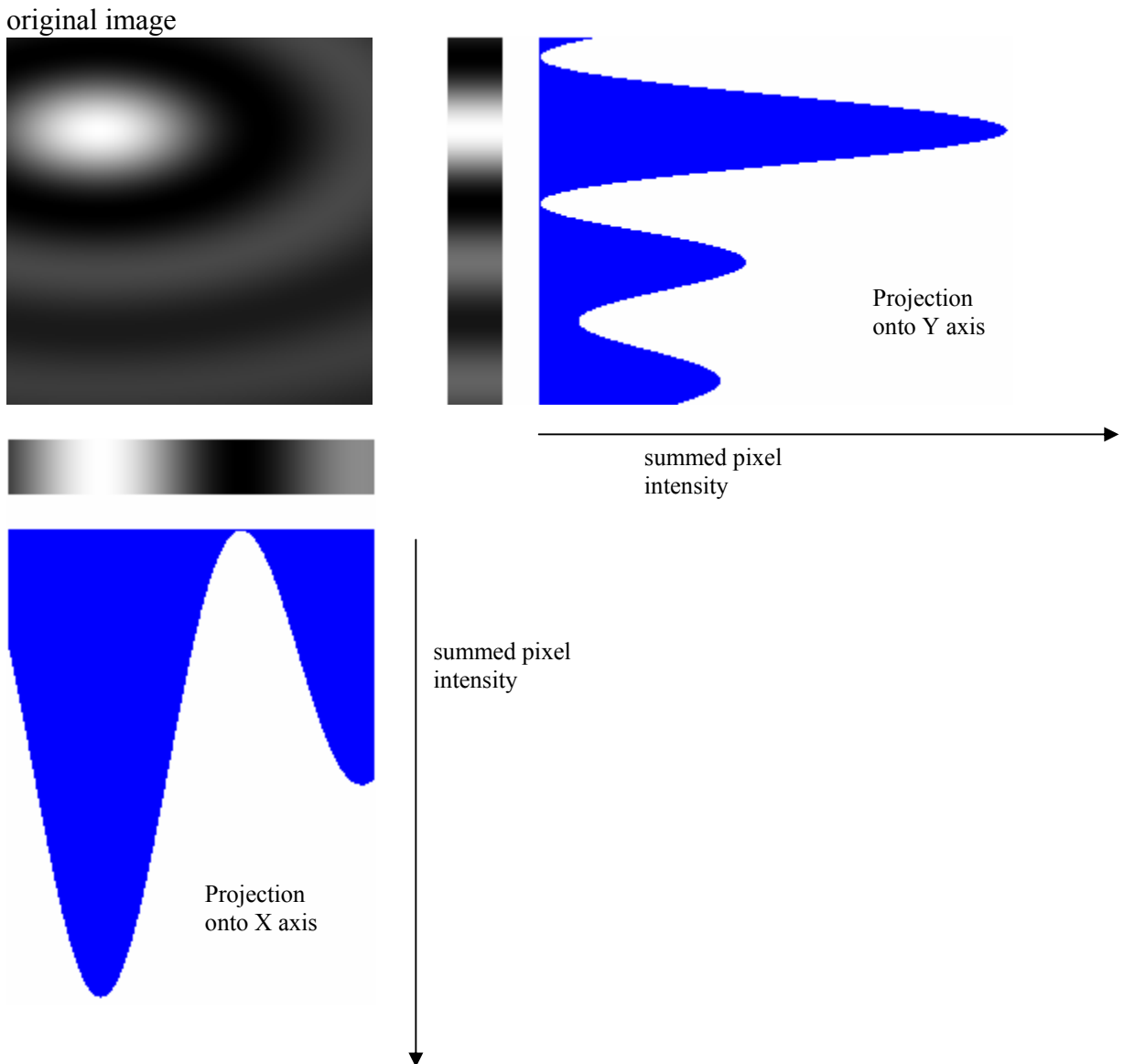
**angle** – The projection angle (in radians), for the case of an arbitrary projection angle. An angle of 0.0 is the same as projecting the image onto the X axis.

**axis** – The axis (either `MV_PROJECTION_ONTO_X_AXIS` or `MV_PROJECTION_ONTO_Y_AXIS`), for the case of a projection onto one of these specific axes.

Results:

The projection image is initialized to hold the projection. The image type (see section 3.1) will always be type `MV_IMAGE_TYPE_DOUBLE`. The projection will be one-dimensional. For projections onto the Y axis, the columns parameter of the projection is set to 1. For arbitrary projections and projections onto the X axis, the rows parameter is set to 1.

Example:



## 12.2) Localizing fiducials

Projections of certain types of image patterns (fiducials) can be used to very accurately localize the position of the fiducial in the image to sub-pixel accuracy. The user must know the approximate location of the fiducial in the image to use this function.

Function syntax:

```
int mv_localize_fiducial(mv_image *image,
                        int x,
                        int y,
                        int columns,
                        int rows,
                        int maximum_deviation,
                        int direction,
                        double *position,
                        double *score);
```

Parameters:

*\*image* – Pointer to variable of type `mv_image`. This holds the image of a fiducial to be localized.

*x, y* – The top-left corner of the rectangular region of the image that is known to hold the fiducial.

*columns, rows* – The size of the rectangular region of the image that is known to hold the fiducial.

*maximum\_deviation* – This is the maximum deviation (in pixels) from the user defined rectangle (that is, the *expected* location of the fiducial) to the *actual* location of the fiducial.

*direction* – This must be either `MV_LOCALIZE_FIDUCIAL_X` or `MV_LOCALIZE_FIDUCIAL_Y`. The fiducial location algorithm applies only to the chosen axis.

*\*position* – Pointer to variable of type `double`. This will be returned with the position of the fiducial centerline along the chosen axis relative to origin of the image (in pixels to sub-pixel accuracy).

\*score – Pointer to variable of type double. This will be the normalized auto-convolution value. It will be 1.0 for a “perfect” fiducial. The user will need to experiment with their particular fiducial to determine a threshold for acceptance (if needed).

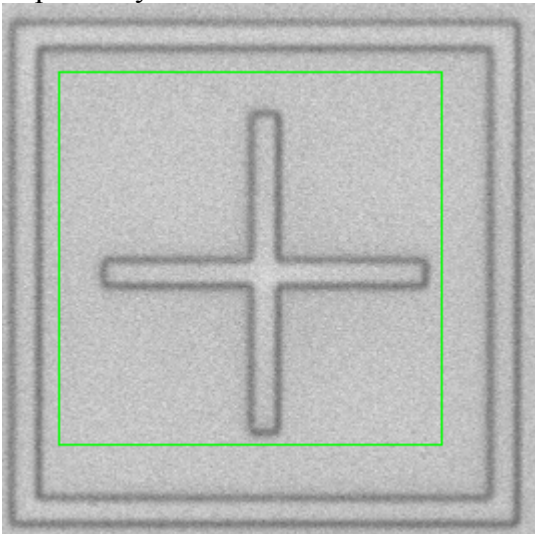
#### Results:

The image is projected onto the specified axis, but only along in the band of the image defined by the rectangle of interest. The rectangle of interest defines the portion of the projection that is autoconvolved with itself. The auto-convolution is centered on the rectangle of interest and is computed for positions +/- the specified maximum deviation from that center.

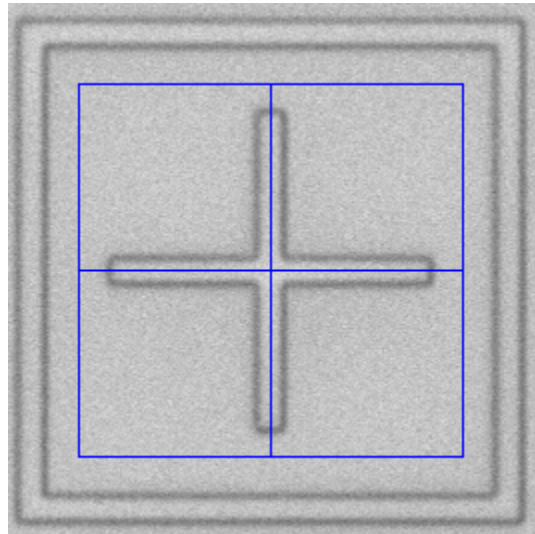
The maximum value found for the autoconvolution is assumed to be the centerline of the fiducial. The position is then interpolated to subpixel accuracy and returned with the normalized score.

#### Example:

User-defined rectangle placed imprecisely over fiducial:



Result of fiducial localization function:



### **13) Miscellaneous functions**

#### **13.1) Version number**

Function syntax:

```
int mv_version(int *version);
```

Parameters:

\*version – Pointer to variable of type int. This is returned holding the version number.

Results:

The version number is a fixed-point decimal number with two digits of fractional value. Hence, for instance, a version value of 102 is used to represent version 1.02 of the MV Lib software.

### 13.2) Image statistics

This function computes some basic statistics on the pixel intensity values in an image. The structure used to hold the statistics is defined below. Note: histograms are computed only for binary and 8-bit images. The median and mode is computed only for 8-bit images.

```
typedef struct mv_statistics
{
    int histogram[256];
    double min;
    double max;
    double average;
    double standard_deviation;
    double mode;
    double median;
} mv_statistics;
```

Function syntax:

```
int mv_image_statistics(mv_image *image,
                       mv_statistics *statistics);
```

Parameters:

*\*image* – Pointer to variable of type *mv\_image*. This holds the image to be used in computing statistics.

*\*statistics* – Pointer to variable of type *mv\_statistics*.

Results:

The statistics for the image are returned in the structure passed to this function.

### 13.3) Scene angle

Many images contain “scenes” that are dominated by parallel lines, or by lines intersecting at right angles. An example would be the scribe lines between die on a silicon wafer.

This function will determine the angle of these predominant lines in the image.

Function syntax:

```
int mv_scene_angle(mv_image *image,
                  int direction_flag,
                  double *angle);
```

Parameters:

**\*image** – Pointer to variable of type `mv_image`. This holds the image on which the scene angle is to be measured.

**direction\_flag** – This must be `MV_SCENE_ANGLE_UNIDIRECTIONAL` or `MV_SCENE_ANGLE_BIDIRECTIONAL`. In the case of the unidirectional flag, the function attempts to find only one set of dominant parallel lines. In the case of the bidirectional flag, the function attempts to find two sets of parallel lines that are assumed to be orthogonal to each other.

**\*angle** – Pointer to variable of type `double`. This variable is used to return the overall scene angle computed for the image.

Results:

The scene angle in radians (from  $-\pi/2$  to  $\pi/2$  for the unidirectional case, or  $-\pi/4$  to  $\pi/4$  for the bidirectional case) is returned.

## 14) Appendix A – Glossary of terms

**8-bit image** – An image in which each pixel intensity value is represented by 8 bits. Thus, the pixel intensity can take on one of  $2^8 = 256$  values.

**binary image** – An image in which each pixel can take on only one of two possible values (0 for black, and 1 for white).

**bit** – Short for “binary digit”. This is the smallest element of information that can be held in a digital system. A bit can be in one of only two states (0 or 1).

**black** – A value (usually 0) representing the darkest pixel value an image can have.

**blob** – A term used to describe a connected region of pixels. Since the region can have very irregular borders and be of genus greater than unity, “blob” is a term to describe a shape that is difficult to quantify otherwise.

**buffer** – A linear array of contiguous memory used to hold a 2-D image.

**clipping** – This term refers to a mathematical process used to keep a pixel intensity value within fixed limits (e.g. 0 to 255 for an 8-bit image). When a pixel value is computed that puts it outside its limits, it is “clipped” to get it back within the limits. All pixel values less than the lower limit are set to the lower limit and all pixel values greater than the upper limit are set to the upper limit.

**complex image** – An image in which each pixel value is represented by a complex number.

**coordinates** – The spatial coordinates of an image dictate how individual pixels are accessed. Most users are familiar with images displayed on a screen (CRT or LCD monitor). Thus, an image has a vertical (up/down the screen) and horizontal (left/right across the screen) coordinate. In MV Lib, access to pixels is via a Cartesian coordinate system in which pixel (0,0) is at the top-left corner of the image, The X component corresponds to the horizontal axis of the screen and increases going from left to right. The Y component corresponds to the vertical axis of the screen and increases going from top to bottom.

**fiducial** – A pattern, or object, within an image specifically placed there to aid in localizing a point or points within the image. Usually fiducials are made to be easy to find and have good contrast and symmetry (for instance, a crosshair).

**grey level** – A discrete (quantized) pixel intensity value.

**histogram** – A linear array of elements (values) in which each element of the array represents a “bin”. Each bin holds the number of pixels of a specific grey level that

occurs in the image that the histogram was computed for. Thus, the number of bins in the histogram must be equal to the number of grey levels in the image. The sum of all bins in the histogram must be equal to the number of pixels in the image.

**image** – A two-dimensional array of pixel values that can be displayed to convey meaning of some sort. In most cases the image formation process consists of a projection of a 3-D scene through an optical system onto a 2-D focal array (such as a CCD sensor).

**intensity** – This is the numeric value assigned to a pixel. Ideally, the intensity value is directly proportional to the number of photons that hit the pixel sensor over the integration period (hence, the intensity of light at the pixel).

**lookup table** – A linear array of values which is indexed by the another set of values. Thus an old value is used as an index into a lookup table to extract a new value. This is the basis on which pixel intensity mapping transformations are performed.

**model** – An image (or portion of an image), either real or simulated, that is used to represent an object that needs to be located in other images, or other portions of the same image.

**NCS** – Short for “normalized correlation search”. An algorithm that uses normalized correlation to locate instances of a model within an image.

**object** – This term is used loosely to define any recognizable feature or group of features within an image. An object within an image usually corresponds to some real-world object of interest (for example, a specific bonding pad in a die on a silicon wafer).

**pixel** – Short form for “picture element”. This is the smallest discrete element in an image.

**resolution** – The term “resolution” can refer to spatial resolution (for example, pixels per millimeter), or intensity resolution (for example, 8 bits (256 grey levels) per pixel). Both types of resolution are critical in machine vision applications and should be considered carefully when developing an application.

**threshold** – A value used to make a decision. For example, when referring to normalized correlation scores, a threshold of 0.75 may be set. Scores at or above the threshold indicate a pass, scores below the threshold indicate failure.

**white** – A value representing the lightest pixel value an image can have. For a binary image, this value is 1. For an 8-bit image, this value is 255.

## 15) Appendix B – Error codes

Errors are encoded in the file “mv\_errors.h”. Application developers should always use the defines for the error codes, never the numerical codes themselves. All possible errors and their likely source are listed below:

`MV_STATUS_SUCCESSFUL_COMPLETION` – This code occurs when a function has been completed successfully. All other error codes indicate errors, this is the only code that indicates the absence of an error.

`MV_ERROR_IMAGE_UNINITIALIZED` – An operation was attempted on an image that was not initialized.

`MV_ERROR_MEMORY_ALLOCATION_FAILURE` – An attempt to allocate memory within a function failed (a NULL pointer was returned). This could be caused by a lack of system resources (too many other images already in memory) or the parameters specified for an operation required an unusually large image size.

`MV_ERROR_PARAMETER_OUT_OF_RANGE` – Range checking is performed on most parameters passed to MV Lib functions. If a parameter is outside of the allowed range, this error code is returned.

`MV_ERROR_IMAGE_TYPE_MISMATCH` – This occurs when trying to perform a function meant for one type of image (see section 3.1) using an different type of image. For example, trying to run `mv_ncs_train()` using an image of type `MV_IMAGE_TYPE_DOUBLE` will result in this error code since the NCS training function requires an 8-bit image.

`MV_ERROR_IMAGE_SIZE_MISMATCH` – Some functions, such as the image algebra functions, require two images that must be the same size. Failure to pass two images of the same size to these functions will result in this error code.

`MV_ERROR_IMAGE_SIZE_TOO_SMALL` – This error code is returned if the image is too small. For example, trying to run a Sobel filter function on a 2x2 image.

`MV_ERROR_LICENSE_EXPIRED` – This will be returned if the license to the MV Lib library is missing or has expired. Lack of a proper license will make this error occur on all MV Lib functions.

`MV_ERROR_CANNOT_OPEN_FILE_FOR_WRITE` – An attempt to write to a disk file failed. Check the file path to make sure it is valid and that the file is not read-only.

`MV_ERROR_CANNOT_OPEN_FILE_FOR_READ` – An attempt to read from a disk file failed. Check the file path to make sure it is valid.

`MV_ERROR_UNIDENTIFIED_FILE_TYPE` – When reading a file from disk, MV Lib uses information in the file header to automatically determine the file format. When the file format cannot be determined, this error code is returned.

`MV_ERROR_UNSUPPORTED_FILE_TYPE` – This occurs when reading a file format from disk that can be identified, but is not supported.

`MV_ERROR_INCOMPATIBLE_FILE_TYPE` – This occurs when trying to write an image to the disk that is incompatible with the file format chosen. For example, an image of type `MV_IMAGE_TYPE_DOUBLE` cannot be written to a GIF file, since the GIF file format only supports 8-bit images.

`MV_ERROR_FILE_CORRUPTED` – When reading data from an identifiable file format, erroneous data was detected.

`MV_ERROR_FILTER_SIZE_TOO_SMALL` – Filters must have dimension 1 or greater in both X and Y. Also, 1x1 filters are not permitted.

`MV_ERROR_FILTER_SIZE_TOO_LARGE` – A user-defined filter is too large. For example, a 5x5 image cannot be processed with a 6x6 filter.

`MV_ERROR_MODEL_SIZE_TOO_LARGE` – Normalized correlation search models can be no larger than 256 pixels in each dimension.

Note: there are other (generic) error codes defined for MV Lib in the “mv\_errors.h” file. These error codes are currently reserved and will not appear when using MV Lib.