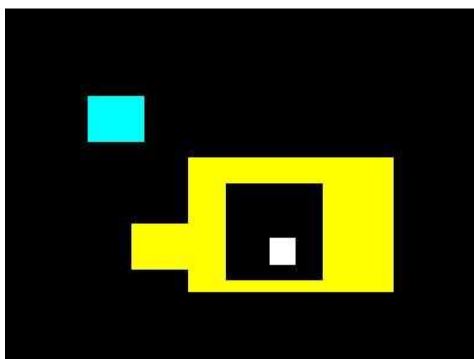
Matlab function reference manual for the PILATUS pixel detector Surface Diffraction Station X04SA Materials Science beamline, Swiss Light Source

written by

P.R. Willmott, R. Herger, C.M. Schlepütz, S.A. Pauli, and D. Martoccia

Swiss Light Source
Paul Scherrer Institut
CH-5232 Villigen



CVS Version `Tag: X_PILATUS_1_1`

December 2006

Contents

1 Introduction **1**

1.1 Software requirements 1

1.2 Conventions 1

2 Atomic functions **3**

2.1 imageread.m 3

2.2 imagewrite.m 4

2.3 medianfilter.m 5

2.4 histmask.m 7

2.5 setroi.m 8

2.6 roimanip.m 10

2.7 surffit.m 12

2.8 fillstack.m 12

2.9 makeffcorr.m 13

3 Beamline-specific functions **15**

3.1 readscanlog.m 15

3.2 fftest.m 16

3.3 imageviewer.m 18

Index **20**

Chapter 1

Introduction

1.1 Software requirements

Most of the software described in this document was created using MATLAB R2006a. In most (if not all) cases, using the older MATLAB release 14 will suffice, though the authors guarantee no backwards compatability.

Most of the functions here require the MATLAB image processing toolbox, while the (as yet incomplete) function `surffit.m` uses the optimization toolbox.

It is stressed that the functions described here and also downloadable from the Surface Diffraction website of the Materials Science beamline at the Swiss Light Source have been developed primarily for in-house applications. The modular nature of these routines was conceived for easy transfer to other applications and needs, so other users can profit, without having to re-invent the wheel. *However, we provide no software support or debugging services, nor can we guarantee the correctness of these routines*. Responsibility for any use or adaptation of these routines therefore lies entirely with the user.

1.2 Conventions

To use this document efficiently, there are some conventions one should be aware of.

1. Matlab code is cited in the `verbatim` style.
2. The output of functions is given in square brackets, e.g., `[output]`.
3. Input arguments are specified in parentheses, e.g., `(input)`.
4. Optional arguments are enclosed by bra-kets, e.g., `<option>`.
5. String variables are held within single quotes, e.g., `'string'`.
6. Scalars or vectors start with a lower case letter, e.g., `dim`.
7. Multidimensional arrays start with a capital letter, e.g., `Im`.

8. Scalar variables in equations are written as a or A.
9. Vectors in equations are bold faced, e.g. $\mathbf{a}$ or $\mathbf{A}$.
10. Matrices in the math mode are written in the caligraphic alphabet, e.g., $\mathcal{A}$.

Note please that by typing `help myfunction.m` in MATLAB, you can get help about the arguments, usage, etc., on that particular function.

Chapter 2

Atomic functions

This chapter describes the following MATLAB functions:

1. `imageread.m` – reads an image (img, tif or tiff, ff)
2. `imagewrite.m` – writes an image (tif or tiff, jpeg or jpg, ps or eps, ff)
3. `medianfilter.m` – performs a median or mean filter in the plane of the image
4. `histmask.m` – produces a mask based on two threshold values
5. `setroi.m` – selects a region of interest within an image
6. `roimanip.m` – manipulates regions of interest within an image
7. `surffit.m` – to be completed
8. `immerge.m` – merges a stack of images
9. `fillstack.m` – fills a stack of images
10. `makeffcorr.m` – creates a flatfield correction file

2.1 imageread.m

The function `imageread.m` reads an image of type `img`, `tif` (or `tiff`), `ff`, or `edf`, and returns a 2-dimensional data matrix.

```
[Im, <header>] = imageread (filename, format, dim, <colorDepth>)
```

- `Im` – returned matrix of dimension `dim`
- `filename` – filename [+ path (absolute or relative)] of the file
- `format` – format of your input file

  - 'img' – tvx .img format
  - 'tif' – tif format with extension .tif
  - 'tiff' – tif format with extension .tiff
  - 'ff' – tvx flatfield tif format (= float tif) with extension .tif
  - 'edf' – ESRF data format with extension .edf
- dim – matrix dimension as vector, e.g., dim = [xdim, ydim]
- colorDepth – Color-depth of the .img file in number of bits (default is 32 bit, for PILATUS 2 images) Values can be 8, 16, 32, or 64 for integer arrays (e.g., 16 must be used for a 16-bit PILATUS 1 image), or use -1 if you want to load floating points of format 'double', e.g., if you want to load an image that is already flatfield corrected.

2.2 imagewrite.m

The function imagewrite.m writes an image to disk. Supported formats are img, tif, and edf. This routine is not meant to create graphical output of pixel data for presentatin purposes (e.g., jpg or eps). The intention is, rather, to write data in a specific data format, with the precision needed (up to 64 bits per pixel are supported) where, in case of tif and edf, you also can store information in the header, e.g., information on the scan or the beamline, etc.

```
[<status>] = imagewrite (Im, filename, format, <colorDepth>)
```

- <status> – Optional output parameter
  - 0 – saving was not successful
  - 1 – saving was successful
  - 2 – file existed before, but was overwritten
- Im – Matrix (= image) of size $(n \times m)$ to save. The elements of the matrix can be floating point numbers
- filename – filename [+ path (absolute or relative)] of the file to be written
- format – format of your output file
  - 'img' – img format (without header) with extension .img
  - 'tif' or 'tiff' – tif format with extension .tif with a header of 4096 bytes
  - 'edf' – edf format as .edf with a header of 1024 bytes (ESRF standard, but neither unique at the ESRF, nor required by the data format itself)
- <colorDepth> – Color-depth of the image in number of bits (default is 32 bit for PILATUS 2 images). Values can be 8, 16, 32, or 64. Note: 16 is required for 16-bit PILATUS 1 images.

2.3 medianfilter.m

The function `medianfilter.m` filters an image in the plane of that image. This is normally required for the substitution of dead or hot pixels. In the first model of the pixel detector (PILATUS I), this facility was essential to obtain reliable data. The second generation model (PILATUS II) has essentially no hot pixels, and typically less than 0.1% dead pixels. Median or mean filtering has therefore become much less critical.

You can either choose a median (the default setting) or a mean filter. In the standard operating mode, the filter only operates on pixels that are more than $n\sigma$ standard deviations away (above and below) from the median or mean of their neighbours. The default value for n (matlab argument `nsigma`) is 5. The number of neighbours around any given pixel is defined by a box ("kernel") of size $(l \times m)$, and is hence equal to $(l \times m) - 1$. Because the median filter selects the em middle value [i.e., the $(l \times m + 1)/2$th element of $l \times m$ elements selected in ascending order], both l and m must be odd integers. The default (and minimum) size of the kernel is (3×3).

The default setting is to median filter a complete image. If you intend to filter only specified pixels from your image, a value of $n = -1$ must be entered for the argument `nsigma` and a mask image of the same dimensions as those of the image being filtered, consisting of a set of zeros and ones, must be provided. Those mask image pixel positions with value 1 are filtered (median or mean).

Internally, the filter creates a 3D-stack, each layer consisting of a circularly shifted image (see Fig. 2.1), i.e., for a given matrix element $\mathcal{A}(i, j)$ all neighbouring elements specified by the kernel (e.g. for $3 \times 3 : \mathcal{A}(i - 1, j - 1) \ldots \mathcal{A}(i + 1, j + 1)$) are stored in the 3D-stack (see Fig. 2.1). The filter (median or mean) and the standard deviation are applied to the third (vertical) dimension of the stack. The selected pixels (be they from a mask filter or by exceeding the $n\sigma$-criterion described above) are then replaced by the median or mean calculated from the stack.

The edge pixels are treated specially. An "edge" pixel is defined as one which has less than $(l - 1)/2$ $[(m - 1)/2]$ pixels between itself and the edge of the image in the x- [y-] direction, and therefore for which the kernel spills over the edge of the image. In order to make filtering possible, the image is symmetrically expanded by replication of the edge pixels. The size of this expansion depends on the kernel size – for an $l \times m(l, m = 3, 5, 7, \ldots)$ kernel, $(l - 1)/2$ pixels are added at both ends in the x-direction and $(m - 1)/2$ pixels in the y-direction. Please note that due to the replication of the border elements, hot pixels in the border region may not be filtered[1].

An associated array of the same dimensions as the image can be optionally output, in which the standard deviations are written. These are calculated from all the neighbouring pixels specified by the kernel, but not the pixel itself, in case it is hot or dead, which would result in very large standard deviations.

```
[ImOut, <SigmaOut>, <noCorrPix>, <CorrPixMask>] = medianfilter (ImIn, <kernel>, <nsigma>,
<SigmaIn>, <Maskimg>, <method>)
```

- `ImOut` – Filtered image
- `<SigmaOut>` – Matrix containing the statistical errors associated with each element of `ImOut`

[1] We tried to circumvent this problem by adding NaN's as extension, but MATLAB's median filter routine cannot deal with them correctly. But since the signal is usually somewhere around the center of the image, this limitation should not have a significant influence.

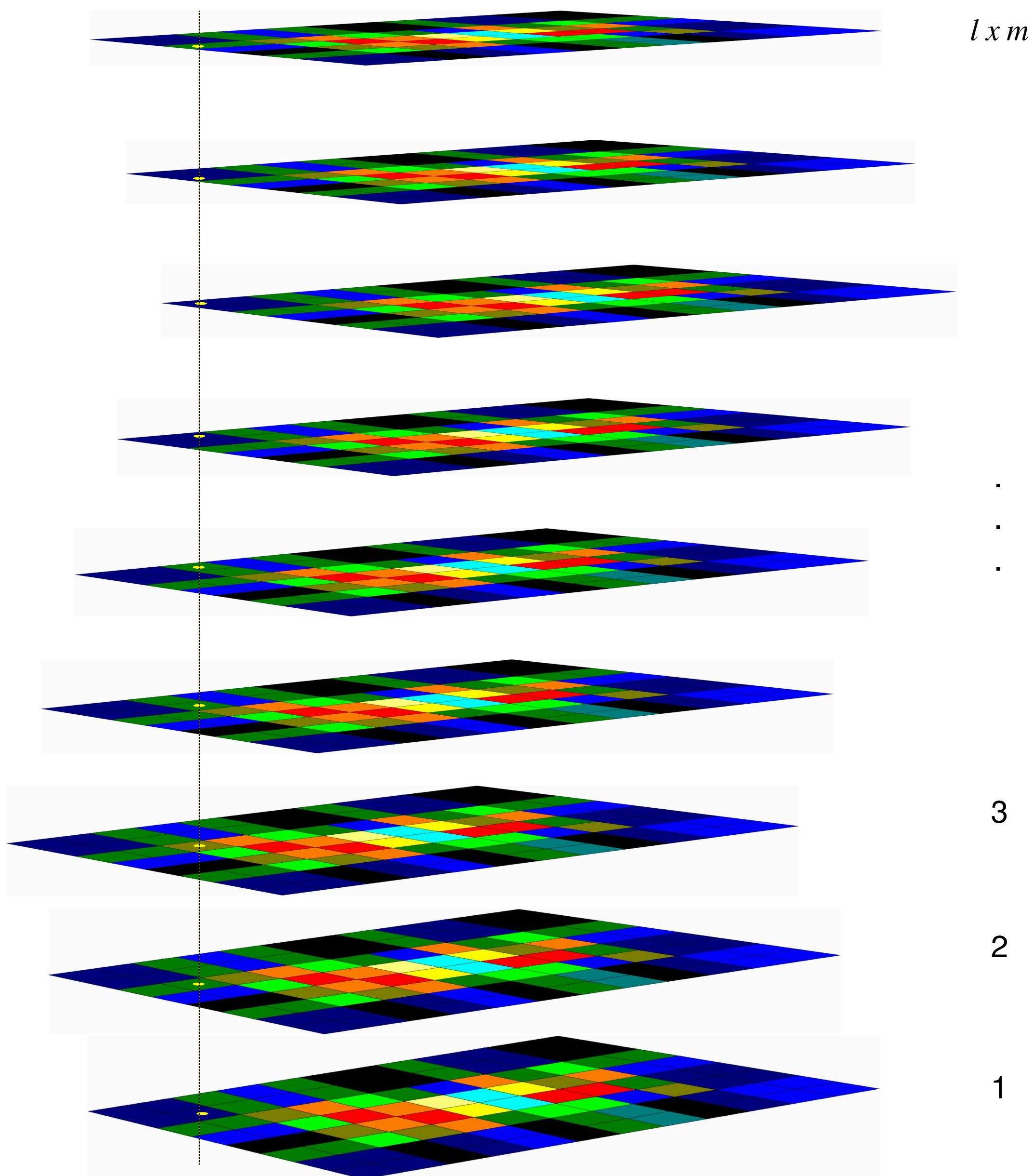


Figure 2.1: A simplified schematic of how the medianfilter works. A kernel of neighbours of size $(l \times m)$ around each selected pixel is defined [here, (3×3)], and a stack of $l \times m$ images high is generated. The position of each image shifted so that every neighbour within the kernel appears once along a vertical line (highlighted here with light yellow spots). Because edge pixels do not have neighbours to all sides, the image is padded out with a border of $(l-1)/2$ in the x-direction and $(m-1)/2$ in the y-direction, in order to permit their filtering. In this example, the (6×10)-pixel image has been padded with a border of width 1 pixel. The intensities of the padding pixels are set equal to the real edge values (Note here that all the pixels in the 1-pixel padding border have the same color as their internal neighbours).

- `<noCorrPix>` – Gives the number of matrix elements (pixels) that have been replaced. This provides information about the quality of the raw image, or, more importantly, the performance of the detector

- `<CorrPixMask>` – Returns a mask (containing either 0s or 1s) of all the modified matrix elements. The number of 1s is equal to `noCorrPix`
- `ImIn` – Image to be filtered
- `<kernel>` – Vector `kernel = [xkernel ykernel]` specifying the amount of neighbouring pixels. Minimum and default `kernel = [3 3]`, only odd numbers allowed for both `xkernel` and `ykernel`
- `<nsigma>` – Pixels are only filtered if they are more than $n\sigma$ standard deviations away from the median or mean within the given kernel, default $n\sigma = 5$.

  Special option: $n\sigma = -1$ replaces each pixel selected by Maskimg with the median or mean value of its neighbours for a given kernel
- `<SigmaIn>` – Matrix containing the errors associated with each element of ImIn
- `<Maskimg>` – If a mask image is supplied, only pixels contained in the mask are filtered by choosing $n\sigma = -1$
- `<method>` – selects the filter
  - `'median'` – median filter (default)
  - `'mean'` – mean filter

The function can be used with several optional arguments, for example:

Example 1: You want to specify a bigger kernel.

```
[MyImOut] = medianfilter (MyImIn, [5 5])
```

Example 2: You want to perform a mean filter.

```
[MyImOut] = medianfilter (MyImIn, [], [], [], [], 'mean')
```

Example 3: You want to know the number of corrected pixels, use a standard kernel, but only filter pixels that are more than 15 standard deviations away from their neighbours. You have to specify also a dummy `SigmaOut`, because MATLAB only allows you to process output arguments sequentially.

```
[MyImOut, MyDummySigmaOut, myNoCorrPix] = medianfilter (MyImIn, [], 15)
```

2.4 histmask.m

The function `histmask.m` produces a mask of an image based on a threshold analysis. It is commonly used to identify bad (dead or hot) pixels in flat field data, though can be used for any image.

As a preparatory step in flatfield analysis, one can take a flatfield image, or indeed a stack of images (see `fillstack.m`), and reshape it into a 1D array using the MATLAB `reshape` function, then perform a histogram on this array using the MATLAB `hist` function. The displayed intensity distribution provides visual information on the quality of the flatfield, from which thresholds for `histmask` can be sensibly chosen.

This function sets all pixels that have counts between a lower and a higher threshold to 1, the others to 0.

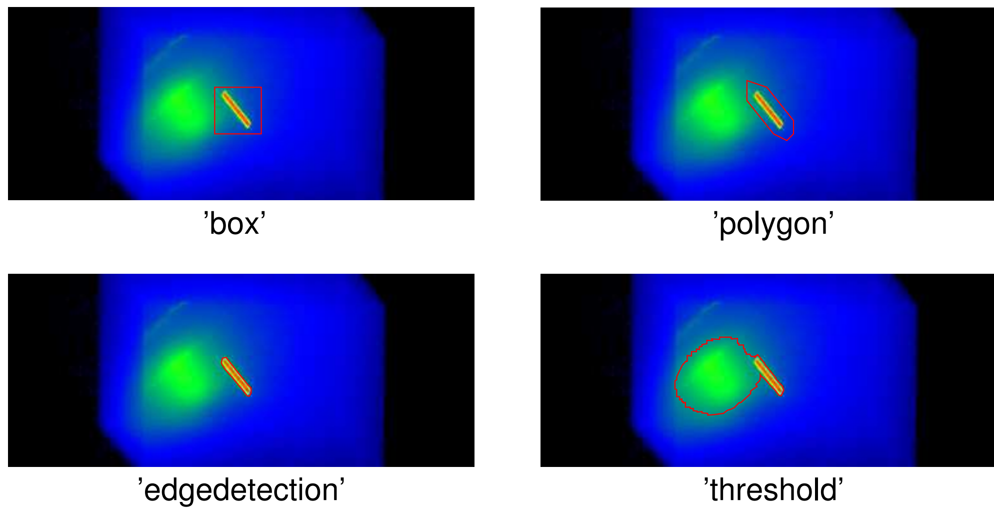


Figure 2.2: The four different methods for defining a region of interest.

```
[Maskimg, <noPix>, <noLowHigh>] = histmask (Im, thresh, <MaskimgIn>)
```

- Maskimg – Mask image for original image Im. The array Maskimg contains
  - 1 – for all pixels between or equal to low and high;
  - 0 – otherwise
- <noPix> – Returns the number of pixels within the thresholds
- <noLowHigh> – Two-element vector containing the number of pixels lower and higher than thresh. The return will be noLowHigh = [Low High]
- Im – Matrix to mask
- <thresh> – Vector containing both the lower (low) and higher (high) threshold, thresh = [low high]
- MaskimgIn – If MaskimgIn is supplied, only pixels contained in the mask are processed, e.g., if you want to analyze a region of interest.
  Note that Im and MaskimgIn have to be of the same size

2.5 setroi.m

setroi.m is a function to select a region of interest (roi) in an image.

```
[BWout, <xi>, <yi>, <nlines>, <outparms>, <method>, <edgedetectionmethod>] = setroi
(I/h, <method>, <parms>, <edgedetectionmethod>)
```

- I/h – 2-D Image data input. If an image I is chosen, the image data is read in and displayed, whereas if h is chosen, the *handle* to an image object inside a figure is called.

- `<method>` – Optional input parameter defining how the roi is created (see Fig. 2.2):
  - `'box'` – Default method, used if this optional argument is not given. If no parameters (`<parms>`) are specified, the user draws a box by clicking on one corner of the desired region and dragging the mouse to the diagonally opposite corner, where a second click completes the definition. `<parms>` and `<outparms>` are 4-element vectors of the form `[xmin ymin width height]`. When using `'box'`, the resulting ROI is guaranteed to be simply connected (i.e., a convex set with only one outside boundary)
  - `'polygon'` – A bounding polygon of arbitrary form. If no parameters are specified, the user can draw the polygon inside the image figure using the mouse. Each vertex is added by a single left-click on the image. To finish the polygon use a double click. This vertex will then automatically join up with the first defined vertex. Care must be taken to avoid intersecting polygon lines, which can produce undesired results. `<parms>` and `<outparms>` are *n*-by-2 arrays with the *x*- and *y*-coordinates of the bounding polygon vertices in the first and second column, respectively. `<outparms>` returns vertices of a closed polygon (i.e., the first *xy*-pair is equal to the last one). If the optional input `<parms>` is not a closed polygon, `setroi` will close it by copying the first vertex to the end of the array.
  - `'edgedetection'` – Finds the ROI by detecting edges (i.e., steep gradients) in the image data. Internally, this method uses the EDGE function provided by MATLAB, followed by a sequence of manipulation steps which improve the chances of finding filled regions rather than just the edges by which they are enclosed. `<parms>` and `<outparms>` are directly passed to and from the EDGE function – please refer to the EDGE documentation in the MATLAB help facility for more information about valid parameter values. If `<parms>` is not supplied or is empty, the parameters for edge detection are determined automatically by EDGE. Several algorithms for edge detection are available in EDGE, all of which are also valid in SETROI: 'sobel', 'prewitt', 'roberts', 'log', 'zerocross', and 'canny' (default). These can be specified through the `<edgedetectionmethod>` optional input argument.
  - `'threshold'` – Image data is transformed into a binary mask by comparison with a threshold level (pixels with intensities above the `threshold` parameter in `<parms>` produce ones, the others zeros). If `<parms>` is not supplied or is empty, the threshold level is determined automatically by the GRAYTHRESH function in MATLAB, otherwise the specified level is used for the conversion. A series of manipulation steps after the thresholding procedure is applied to remove noise and to produce more reliably filled regions which can be used as sensible ROIs.
- `<parms>` – Optional input defining the parameters for the chosen ROI-defining method. Refer to the descriptions of the different methods above. (default = [], resulting in user interaction for methods 'box' and 'polygon' and automatic parameter adjustments with methods 'edgedetection' and 'threshold').
- `<edgedetectionmethod>` – Optional input string parameter only needed when specifying the algorithm when using the method 'edgedetection'. The 6 different possibilities are listed above in the description of `'edgedetection'`

- `BWout` – The logical mask output image of the same dimensions as `I`, with zeros outside and ones inside the selected region. The original image is multiplied by the mask, which effectively sets everything outside the ROI to zero.
- `<xi>, <yi>` – Returns the x- and y-coordinates of the bounding polygon(s) for the region(s) in `BWout`. If the region in `BWout` is simply connected (i.e., a convex set with only one outside boundary), `xi` and `yi` are vectors of size $1 times N$, where N is the number of vertices on the bounding polygon. If several distinct bounding polygons are present, `xi` and `yi` are returned as $M \times 1$ cell arrays, whereby M is the number of distinct boundaries. Each cell in `xi` and `yi` contains a vector of size $1 \times N_i$ with the x- and y-coordinates of the N_i vertices of the i-th bounding polygon.
- `<nlines>` – Returns the number of distinct boundary polygons present in BWout. For a simply connected set, `<nlines>` is therefore 1 and `xi` and `yi` are returned as simple vectors (see above). For multiply connected sets, `nlines` is greater than 1 and `xi` and `yi` are cell arrays of size `nlines` $\times 1$ (see above).
- `<outparms>` – Returns paramters used by `<method>` in the same format as the corresponding input argument in `<parms>`, meaning that they can be used directly with the next function call to SETROI.
- —¡method¿— – Returns the string name of the method used in SETROI
- —¡edgedetectionmethod¿— – Returns the string name of the edge detection algorithm used in SETROI, but only if the chosen `method` was 'edgedetection'.

2.6 roimanip.m

This function performs a palette of manipulations on the size, position, and orientation of a region-of-interest mask. This function uses the atomic function `boundingpolygon.m`, which is itself based on the MATLAB function `countourc`, used to determine countour lines around 2-D features.

```
[BWout, <xi>, <yi>] = roimanip (BWin, <'method'>, <par>)
```

- `BWin` – The input ROI mask image to be modified, of the same size as the image that is being processed. You must convert the binary image into a label matrix before calling `roimanip`. There are two common ways to convert a binary image into a label matrix in MATLAB:

  a) Using the bwlabel function
  L = bwlabel(BW);

  b) Using the double function
  L = double(BW);

  Note, however, that these functions produce different but equally valid label matrices from the same binary image. Consider FIg. 2.3. The two top images are how "bwlabel" and "double" interpret the original image shown on the left of the figure. The output of double(BWin) is interpreted as one single (segmented) region, while the result of bwlabel(BWin) is interpreted as an image containing multiple (here, three) regions which are to be processed individually and are labelled differently (signified here by the three different colors).

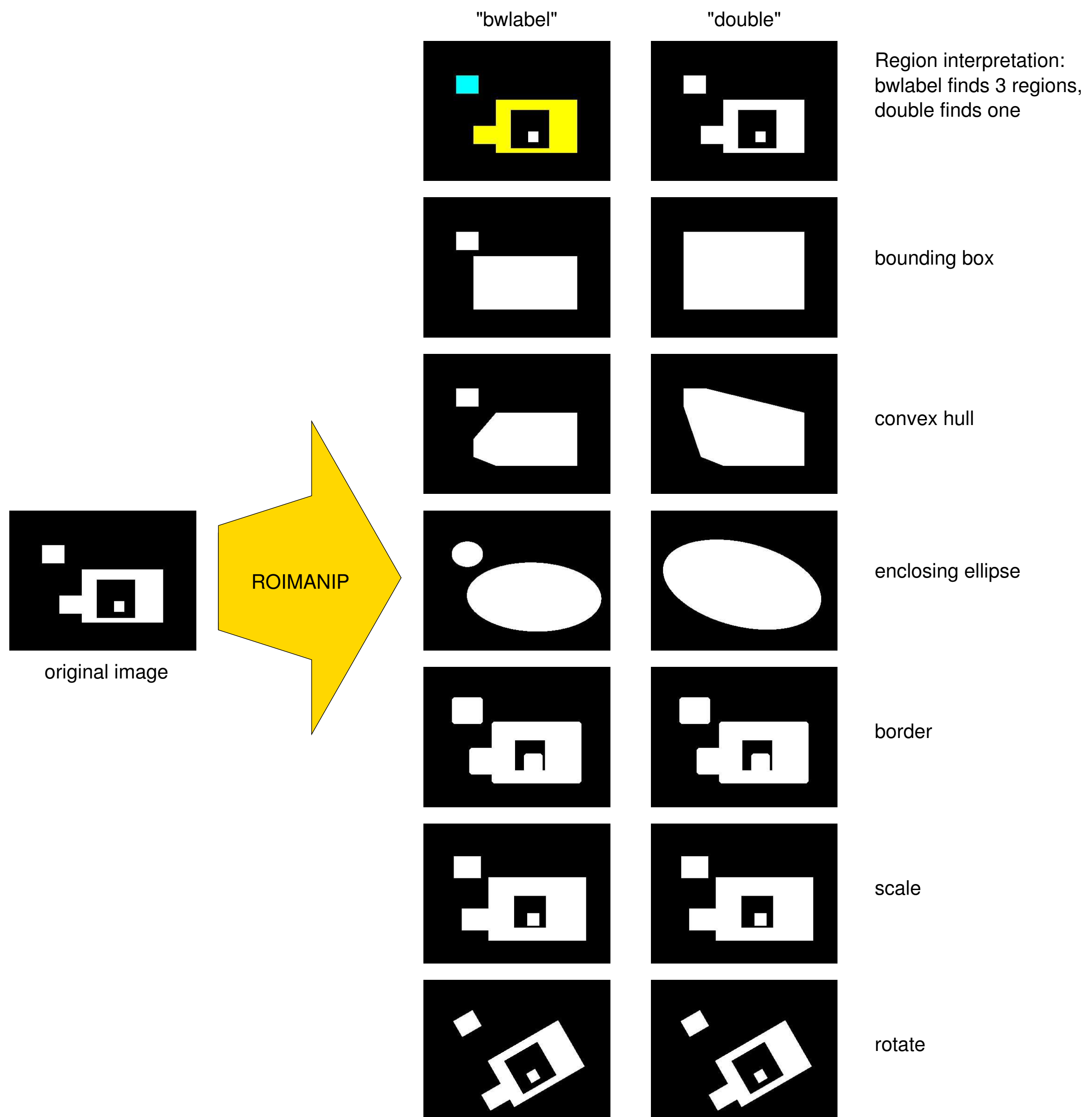


Figure 2.3: Schematic diagram showing how a region-of-interest mask (the "original image" on the left, also referred to as `BWin`) is processed by region-of-interest manipulations. According to whether the mask has been processed using `bwlabel` or `double`, the resulting change in the ROI can be different.

- `<'method'>` – If no method is given, `roimanip` performs the default modification of adding a 10-pixel wide border around the input mask image `BWin`, and returning the resulting mask image in `BWout`. Otherwise, one of the following methods can be given (see Fig. 2.3):
  - `'boundingBox'` – finds the tightest bounding box around each region in BWin and scales it by a factor `<par>` with respect to its center-of-mass coordinates.
  - `'convexHull'` – finds the smallest convex enclosing polygon around each region in BWin and scales

it by a factor <par> with respect to its center-of-mass coordinates.

- 'enclosingEllipse' – finds the smallest enclosing ellipse containing all the points for each region in BWin and scales it by a factor <par> with respect to its center-of-mass coordinates.
- 'border' – moves the border of each region outwards by <par> pixels. Negative values of <par> move the border inwards. No scaling is applied. Sharp corners are rounded by this operation.
- 'scale' – scales each region in BWin by a factor <par> with respect to its center-of-mass coordinates. <par> must be positive. Values smaller than 1 shrink the regions.
- 'rotate' – rotates each region in BWin anti-clockwise by an angle <par> with respect to its center of mass. <par> is given in degrees.

- <par> – parameter which controls the manipulation process.

  For <method> = 'border', <par> specifies the number of pixels by which the border is moved outwards (inwards for negative values). Default = 10.

  For <method> = 'rotate', <par> specifies the anti-clockwise rotation angle in degrees. Default = 5.

  For all other methods, <par> controls the amount of scaling which is applied to the regions with respect to their center-of-mass coordinate. If <method> = 'scale', the default value of <par> is 1.2. For <method> = 'boundingBox', 'convexHull', and 'enclosingEllipse', no scaling is applied by default (i.e., <par> = 1).

- BWout – Binary mask image (of type logical, not a label matrix) with the modified region of interest.

- <xi>, <yi> – Coordinates of the bounding polygons of the regions in BWout. If more than one distinct bounding polygon is present (for example in segmented images, or regions containing holes), xi and yi are cell arrays of size *n*-by-1, where each cell contains the coordinates for one of the *n* bounding polygons. For only one polygon, xi and yi are directly returned as standard vectors (1-by-*k* arrays). To obtain the *i*-th bounding polygon coordinates, use the following syntax:

  ```
  x = xii, y = yii
  ```

  which returns the two vectors x and y of size 1-by-*k*, where *k* is the number of vertices of the *i*-th bounding polygon.

2.7 surffit.m

This function will provide 2-dimensional fitting of backgrounds for subtraction purposes. To be completed

2.8 fillstack.m

fillstack.m is a simple function to create a 3D stack of images. It uses imageread.m to read an image from the disk. On a PSI test machine (pc4095), fillstack.m needed about 20 s to read 100 PILATUS 2 images

(consisting of 32 bit pixels). This function is very useful for handling sets of flatfield images (e.g., evaluating their statistics), or performing averaging/statistical functions in the "vertical" direction (i.e., perpendicular to the planes of the individual images), which is particularly important if your detector has a significant number of "hot" or "unreliable" pixels.

```
[ImStack] = fillstack (path, filename, format, images, dim, <colorDepth>)
```

- `path` – String containing the filepath
- `filename` – String containing the filename without image number digits (e.g., without `00173` in the filename `image00173`)
- `format` – Can be one of the following 5 string inputs
  - `'img'` – tvx .img format
  - `'tif'` – tif format with extension .tif
  - `'tiff'` – tif format with extension .tiff
  - `'edf'` – esrf data format with extension .edf
  - `'ff'` – tvx flatfield tif format (= float tif) with extension `.tif`. This is the format of the output of a flatfield generation produced, e.g., by `makeffcorr.m`. It is *not* the format used for the input into `makeffcorr.m` or `fillstack.m`. They use either `.img` or .tif extensions.
- `images` – Integer vector of the format `images = [start end]`, giving the "from" and "to" image numbers, e.g., `[12115 12619]`
- `dim` – Matrix dimension as vector, i.e., `dim = [xdim, ydim]`, which for the PILATUS 1 is `[366 157]` and PILATUS 2 `[487 195]`
- `<colorDepth>` – Color depth of the .img file in number of bits (default 32). `colorDepth` can be
  - `8`,
  - `16` (for PILATUS 1 images),
  - `32` (for PILATUS 2 images), or
  - `64`.
- `ImStack` – 3D output array containing the images having dimensions `xdim`, `ydim`, `start - end + 1`

2.9 makeffcorr.m

The function `makeffcorr.m` creates a flatfield correction image (2-D normalization file) for the PILATUS 2 detector to correct for the different sensitivities of the individual pixels. `makeffcorr.m` takes the median along the third dimension of the input stack and calculates the standard deviation for each median merged pixel. Each

pixel of the median-merged and error matrices are normalized to their in-plane mean, (but omitting zeros), as follows:

$$\mathcal{F} = \frac{\overline{I}}{I}, \tag{2.1}$$

where $\mathcal{F}$ and I are the resulting flatfield correction and median-merged input matrices, respectively, and $\overline{I}$ is the in-plane mean of I.

Note that `makeffcorr.m` does not perform any histograms or other statistical routines to get rid of hot or unreliable pixels. In order to do so, you have to supply an appropriate `MaskImg`, for example created using `histmask.m`.

```
[Im, <Sigma>] = makeffcorr (ImStack, <MaskImg>)
```

- `Im` – Flatfield correction image, normalized around 1 to correct a pixel's different quantum efficiency for incoming photons.
- `Sigma` – Standard deviation of each pixel of `Im` (ideally, this reflects the counting statistics of each pixel, normalized to the mean of `Im`, i.e., $\sqrt{n_{i,j}}/\overline{n}$.
- `ImStack` – 3D stack of all measured flatfield images.
- `MaskImg` – Binary mask to select the pixels on which `makeffcorr.m` acts.

Chapter 3

Beamline-specific functions

This chapter describes functions that have been specifically developed for use with the in-house research program of the X04SA Materials Science beamline surface diffraction station, and are also likely to be of use for many of the external users. These include the following MATLAB functions:

1. `readscanlog.m` – reads the header and data of a scan log file
2. `fftest.m` – performs statistical analysis on a stack of flatfield images
3. `imageviewer.m` – an interactive GUI for viewing images

3.1 readscanlog.m

`readscanlog.m` is a function to read the scanlog files of the Materials Science Beamline (MSBL). These files are routinely created for each `hklscan` performed with SPEC. The scanlog files contain the most important information of the scan. The exact contents of the file depends on when it was created – early scans produced log files without the scan point numbers or corrected intensity, which are included in later versions. In other words, the program is sufficiently flexible to handle the historical developments of the data thought necessary to save. The log files can contain the following data: scan points, image numbers, monitor intensity I_0 (labelled as "Opto"), raw intensity of the region of interest, intensity corrected for exposure times and filter settings, exposure time, filter transmission, the Miller indices (h,k,l) for both surface and bulk coordinates, important motor positions, and a timestamp. Hence, according to when the scan was recorded, the number of columns containing the data may vary.

`readscanlog.m` returns two arrays. The `Data` array is a 2D matrix containing all the data, stored with double precision. The `header` is a string array of the header information. It has the same number of columns as `Data` and is used to assign the columns of `Data`.

The last six columns of `Data` contain the time information. The timestamp of the scanlog file is split up to a MATLAB date vector of the form
`[yyyy, mm, dd, HH, MM, SS]` in order to simplify calculations. Often for a set of *hklscans*, we record a

reference peak after each scan to monitor possible radiation damage. You can use the timestamp information of your scanlog file to calculate the time dependence of the decrease in intensity of the reference peak, if radiation damage has a significant influence. This date vector can be treated with the standard MATLAB date functions, e.g. `datestr`, `datenum`, and `datevec`. To return to a more user-friendly time information, you can use something like
`mydates = datestr(datenum(Data(:,15:20)), 'yyyy-mm-dd HH:MM:SS').`

```
[Data, header] = readscanlog (filename)
```

- `filename` – Filename (+ path (absolute or relative)) of the scanlog file to be read as string (i.e., surrounded by single quotes (')).
- `Data` – Matrix containing the data of the scanlog file. All numbers are saved as doubles.
- `header` – Array of strings that contain the header information. `header` has the same number of columns as `Data` and correspond to the values in `Data`.

All variables are mandatory.

3.2 fftest.m

`fftest.m` is a function for generating statistical information about flatfield data sets. It is user-interactive (i.e., the user is prompted as the analysis progresses).

```
[Im, <sigma> ,<info>] = fftest (ImStack, <fileOut>, <printer>)
```

- `ImStack` – Flatfield data in the form of a stack of images, created by `fillstack.m`.
- `<fileOut>` – Optional string output "mat" file, containing the fields `ff_dat.ff_mask` and `ff_dat.ff_err`, corresponding to `Im` and `<sigma>`, respectively. The default name of this file if this optional variable is not given is `ff_corr.mat`.
- `<printer>` – Optional string argument specifying the printer for the print-out of the statistical analysis. Default is WBBA˙005˙1.
- `Im` – Output flatfield correction image, normalized to unity.
- `<sigma>` – Optional output image showing the standard deviation for each pixel, normalized to the mean value of `Im`.
- `<info>` – Optional output argument containing information on the image stack `ImStack`, including the mean value, the lower and upper acceptable counting limits, the tolerance factor in percent, and number of bad pixels, as defined by counting limits and tolerance factor.

Some guidelines to interaction with the program: The first action it does is generate a histogram of the counting rate of all the pixels within the stacked images produced by `fillstack.m`, plus (for purposes of comparison) a theoretical Poisson distribution for a detector with a completely homogeneous counting efficiency and

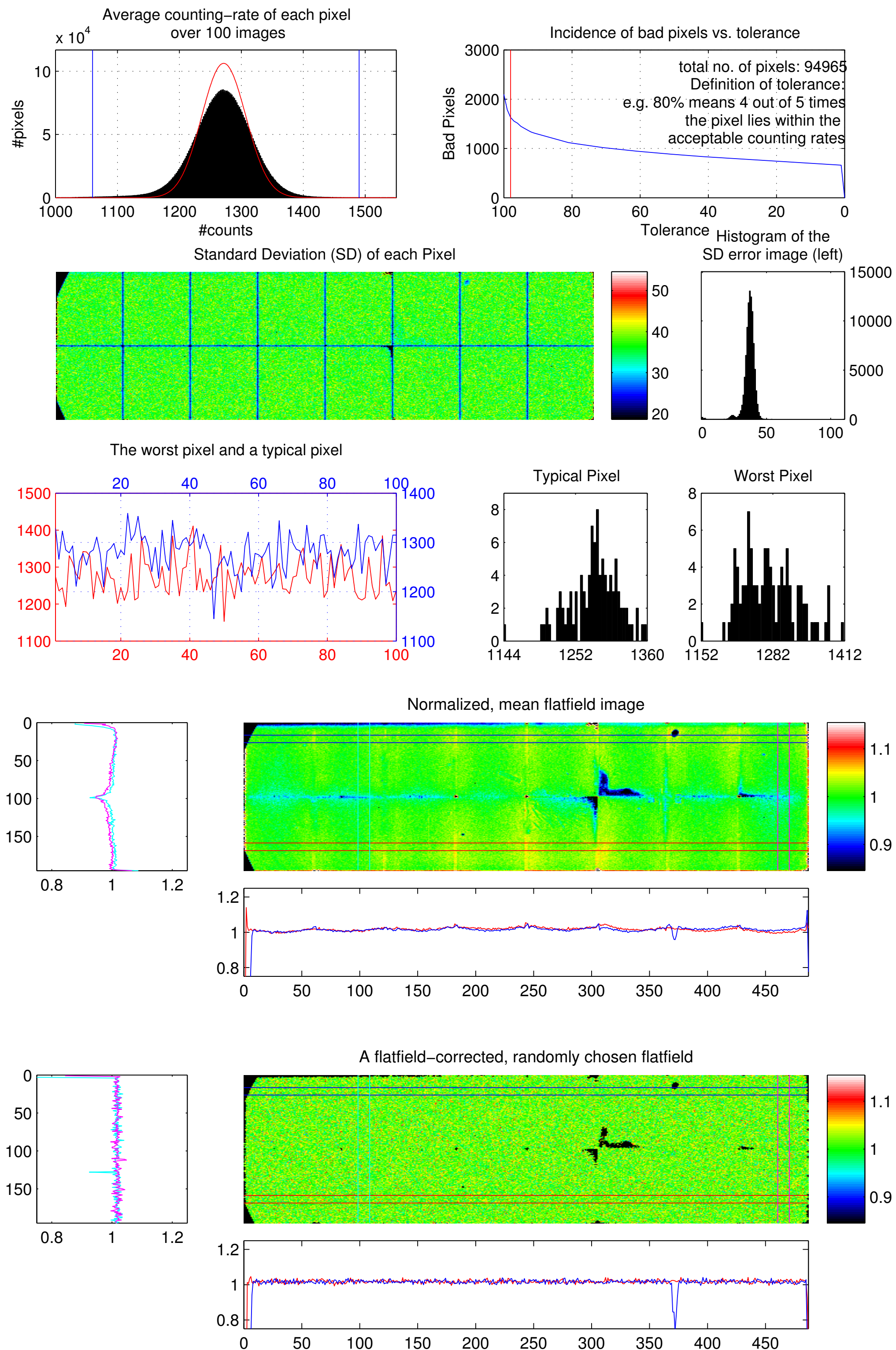


Figure 3.1: Typical printouts generated by `fftest.m`.

an average counting rate equal to that found for the experimental data. Hence for Pilatus II, and 100 flatfield images, a total of $195 \times 487 \times 100 = 9\,496\,500$ pixels are included in the histogram. The program then offers the user to change the $x-$ and $y-$axis limits. It suggests ± 6 standard deviations as the upper and lower limits for counts which are to be considered as being acceptable (i.e., over 99.9% of a perfect Poisson distribution). These can be changed to suit the user. Based on these limits, it then calculates a so-called "tolerance graph". The quantity "tolerance" (perhaps better named "intolerance") is defined in percent as the fraction of the time any given pixel lies within the user-defined acceptable limits. Hence, if a value of 90% is entered, those pixels which register counts outside the limits more than 10% of the time (so for 100 flatfield images, for more than ten of those images), are deemed to be "bad" and are set to zero in the flatfield image file.

Once the tolerance value has been entered, a set of statistical output graphs are generated, and the user can plot this out and/or generate an eps file of it. Then a second side of statistics are generated, showing the mean and normalized image of the original images within the 3-D stack, plus a randomly selected flatfield image from the stack, corrected using the final flatfield correction mask. For both of these images, typical spatial fluctuations along the short and long axes are plotted. This second side can also be printed or saved as an eps file.

3.3 imageviewer.m

This is a graphical user interface (GUI) for interactive image browsing, and is the matlab file for `imageviewer.fig` (see Fig. 3.2). It uses several other atomic functions described in this manual, including `imageread.m`, `medianfilter.m`, `pilatus.m`, `setroi.m`, and `readscanlog.m`, which must all be available for `imageviewer` to run properly. Also, `imageviewer` was developed and tested using the version MATLAB R2006a. Backwards compatibility is neither expected nor guaranteed.

The GUI is to a large extent self-explanatory. Two points worth noting are (1) the (x, y)-coordinates and intensity of any given pixel can be determined by moving the mouse pointer across the "Zoom" window (but not in the "Image Overview" window); and (2) the image is presented such that the photons are impinging from above (i.e., it is viewed from upstream).

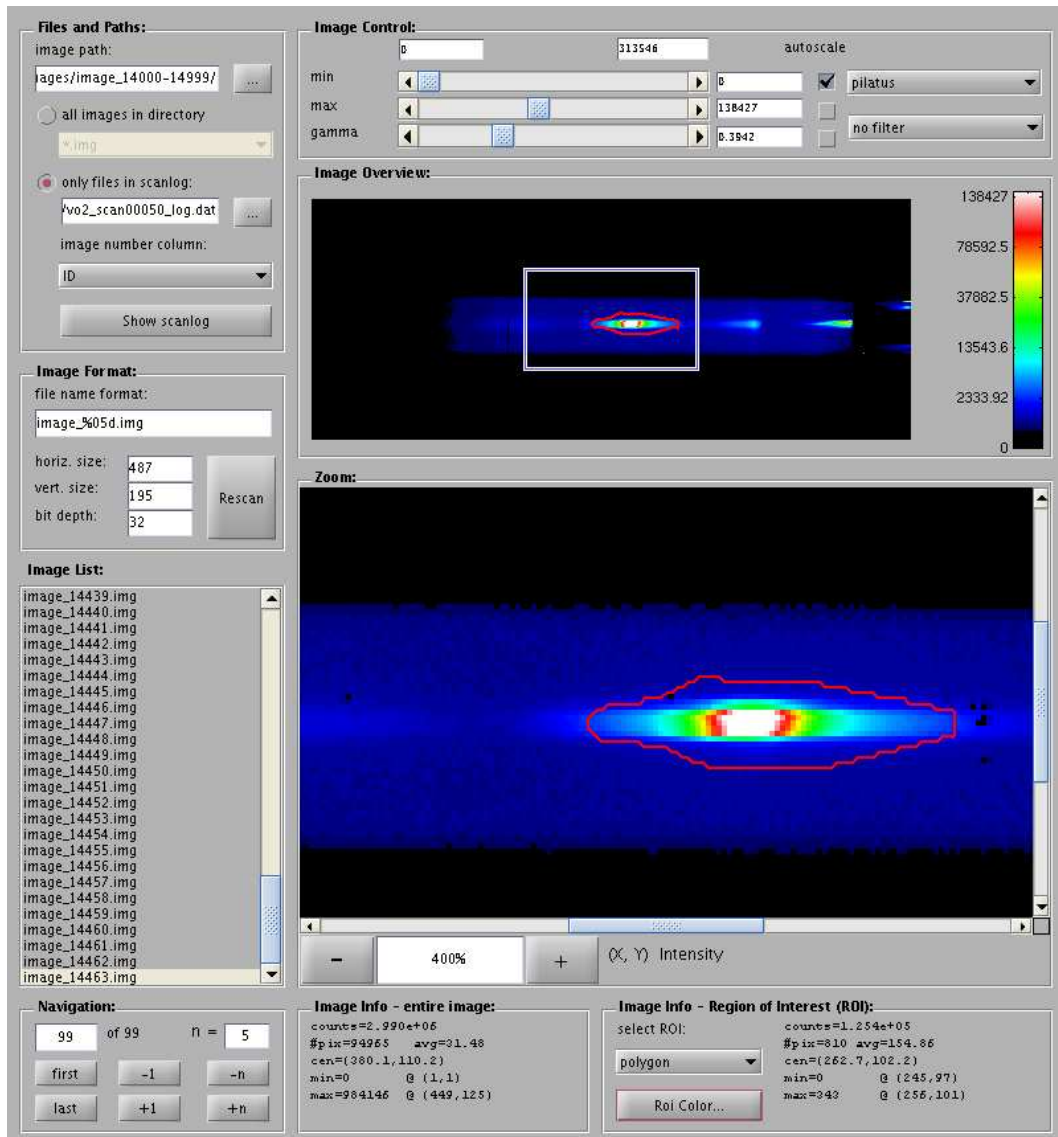


Figure 3.2: A screenshot of the `imageviewer` GUI.

Index

fftest.m , 16

fillstack.m , 13

histmask.m , 7

imageread.m , 3

imagewrite.m , 4

makeffcorr.m , 14

medianfilter.m , 5, 7

readscanlog.m , 16

roimanip.m , 10

setroi.m , 8

fftest.m , 16

imageviewer.m , 18

readscanlog.m , 15