

- **AnalLS**

AnalLS analyses linescan data generated with ScanImage. These are usually present as image stacks, only that lines in an image represent subsequent recordings of the same line.

**Note:** Required include files are:

LoadScanImage.ipf

See also: [LoadScanImage](#)

**AnalLS** (*LWave, start, stop*)

This operation extracts the average brightness of a one-dimensional ROI over time (if the data has been generated in ScanImage and loaded using the [LoadScanImage](#) function) or frame number. *Start* and *stop* are the left and right pixels (!) of the ROI. The resulting wave will be named `nameofwave(LWave)+"_LS"` and contain the normalised fluorescence,  $\Delta F/F$ .

**CallAnalLS** ()

This operation calls [AnalLS](#) and prompts the user for parameters. This function is useful to be placed in a menu for quick access.

- **AutoRegistration**

Use this operation to call [RegisterStack](#) on multiple files in a folder. This is useful for registering a large number of image stacks overnight or while away from the computer. Call `AutoRegistration()` and specify one or many .tiff files. These will be sequentially loaded, registered and saved as Igor binary files in the origin folder.

**Note:** Required include files are:

LoadScanImage.ipf

RegisterStack

- **CenterOfMass**

The functions in CenterOfMass are used for calculating centers of mass of regions of interest based on brightness, displaying them and measuring distances between curves (waves) and points.

**CenterOfMass** (*image,ROI*)

This operation calculates the center of mass for ROIs based on the intensity of *image*. The ROI wave *ROI* must be a [MultiROI ROI wave](#). The result will be stored in the two-dimensional wave `CoM`. `CoM[][0]` contains the scaled X data, `CoM[][1]` contains the scaled Y data. The precision of `CoM` is higher than that of pixels in *image*.

See also: [MultiROI ROI wave](#)

**CoMDistances** (*CoM, [index]*)

This operation calculates the (scaled) distances between all CoMs in wave *CoM* and returns the result as the matrix *DistanceMatrix*. The optional parameter *index* can be used to specify the layer of a compound CoM wave (as returned by the operation [ComByLayer](#)).

### **DistanceMatrix2Column** (*DistanceMatrix*, [*index*])

This operation puts all distances of the distancematrix *DistanceMatrix* in a single column of the wave *DM2C*, so that these data can be used for a histogram, clustering, etc. If the optional parameter *index* = 1, the output wave, *DM2C*, has three columns, the first one holds the distances between two traces, the second and the third hold the index of the traces that the distance was measured between.

See also: [HiClu](#), [HiClu2D](#), [HiCluCP](#)

### **GeometricCenter** (*ROI*)

This operation calculates the geometric center for ROIs in the [MultiROI ROI wave](#) *ROI*. The result will be stored in the two-dimensional wave *GeoC*. *GeoC*[[0]] contains the scaled X data, *GeoC*[[1]] contains the scaled Y data. The precision of CoM is higher than that of pixels in *ROI*.

### **ShortestDistance** (*func*, *relX*, *relY*)

This function returns the shortest distance between a function *func* (1D wave) and a point, defined by *relX* and *relY*. The wave has to be scaled, as have the points.

See also: [ShortestDistance2](#)

### **ShortestDistance2** (*xfunc*, *yfunc*, *relX*, *relY*)

This function returns the shortest distance between a curve, defined by *xfunc* and *yfunc* and a point, defined by *relX* and *relY*. Waves and points have to be scaled.

See also: [ShortestDistance](#)

### **DrawCoMNumbers** (*CoMWave*, *bgimg*, *WindowName*, [*fontsize*])

This function draws the ROI numbers as stored in *COMWave* at the position of the center of mass in the window *WindowName*. *bgimg* is needed to get the dimension of the background image, which should be displayed in *WindowName*. *FontSize* is optional and useful for very large images, as fonts don't scale when the image size is changed.

**Note:** Numbers are drawn relative to the borders of the window, not the image. Therefore changing the axes won't affect the drawn numbers. Also note that the origin of the image is expected to be in the lower left. If the image is to be displayed with the origin in the upper left, the function has to be adjusted accordingly. A comment in the function points out where this has to be done.

Related Topics: [ResultsByCoef](#)

- **Customcolor**

**rainbowLUT** (*levels, outputwave*)

This function generates a rainbow lookup table with a number of different colours equal to *levels*. *Outputwave* will be the name of the so generated lookup table. This function is called automatically in [QuickAnal](#) in order to generate as many colours as there are regions of interest.

Related Topics: [ModifyImage](#)

- **Difference**

The functions in Difference are used to calculate a "difference image", i.e. subtract a baseline image from a response image. Usually, frames over a specified time interval are averaged to generate the baseline and response images. These functions are called from the ImageAnalysis Control Panel, when "Response" is selected as the thresholding method.

**RunDifference** (*stack3D, startbaselineF, stopbaselineF, startresponseF, stopresponseF*)

This function calculates the difference image by averaging frames in *stack3D* from *startbaselineF* to *stopbaselineF* to calculate the baseline image and by averaging frames from *startresponseF* to *stopresponseF* to calculate the response image. The resulting image is named `NameOfWave(stack3D)+"_RES"`.

**GetTimes** (*stack3d, name*)

This function calls prompts to enter start and stop times and then calls `RunDifference` using these variables.

**Note:** Required include file:  
z-project.ipf

- **dif\_image**

The functions in `dif_image` are used to generate the 2nd derivative of grayscale images and threshold these.

**dif\_image** (*image, [targetname]*)

This function calculates the 2nd derivative of *image* in the x and y axis and sums them. The optional parameter *targetname* is the name of the so calculated image. If left out, the name will be `nameofwave(image)+"_dif"`.

**dif\_image3D** (*image, [targetname]*)

This function calculates the 2nd derivative of the volume data *image* in the x, y and z axis and sums them. The optional parameter *targetname* is the name of the so calculated image. If left out, the name will be `nameofwave(image)+"_dif"`.

**difloop** (*imagestack, [targetname]*)

This function calculates the 2nd derivative of *imagestack* in the x and y axis and sums them for each frame. The optional parameter *targetname* is the name of the so calculated image. If left out, the name will be `nameofwave(image)+"_dif"`.

**mod\_img** (*image*, *i\_limit*, [*targetname*])

This function performs (negative) thresholding on *image*:

First, pixels with a value larger than *-i\_limit* will be set 1 (outside ROI), then those with a value  $\leq 0$  will be set 0 (inside ROI.) The optional parameter *targetname* is the name of the so calculated image. If left out, the name will be `nameofwave(image)+"_mod"`.

**Note:** The threshold, *i\_limit*, is not normalized to *image* within this function. When `mod_img` is called from [ManThresh](#), however, the input to *i\_limit* is first multiplied by the standard deviation of *image* so that similar levels will yield similar results independent of absolute values in *image*.

**xydif\_image** (*image*, [*targetname*])

This function calculates the 1st derivative of *image* in the x and y axis, and thus returns two images. The optional parameter *targetname* is the name of the so calculated images which "\_x" and "\_y" appended to their names. If left out, the names will be `nameofwave(image)+"_difx"` and `nameofwave(image)+"_dify"`. This function is not called in automated analysis, but left for completeness.

- **EqualizeScaling**

These functions quickly transfer scaling or size from one wave to another, even if the numbers of dimensions mismatch.

**CopyScaling** (*source*, *destination*)

This function applies point-by-point scaling from *source* in all 4 dimensions plus the data full scale to *destination* and copies the wavenote. Source and destination don't need to have the same number of dimensions. If *source*, for instance, has 3 dimensions, but *destination* only 2, then those 2 will be appropriately scaled.

**Note:** Dimension labels are not transferred.

**CopySize** (*source*, *destination*)

This function applies the length of the respective axes from *source* to *destination*, regardless of the number of points. It also copies the data full scale and the wave note. This is useful, for instance, after resampling or pixelating an image.

Related Topics: [SetScale](#), [ImageInterpolate](#)

- **ExpDataBase2**

These functions combine [PopulationWaves](#) and associated results into a database and allow extraction of data from a database. Version 1 operated by an entirely different principle and was never adopted by the community.

Databases can be built by launching the control panel, [EDB2\\_CP](#). Information can be extracted using custom functions. A template is given in [ExpDB2\\_Extraction](#).

The [PopWaveBrowser3](#) allows browsing of databases and viewing/changing associated information.

Data is stored by adding a "footer" to the end of a PopulationWave that stores associated information as well as scaling, name of origin, etc., making use of Igor's [Dimension Labels](#). Waves with different numbers of points can be combined, the resulting database will leave the appropriate fields of waves with less points empty (i.e. contain NaN). This allows analysis of the associated information in respect to the original traces, or other associated information. For instance, one can compare traces obtained at different ages, or check whether there is correlation of age with size (see figure).

RO Label		Time		
Row		c0214e003_DB	c0214e003_DB	c0214e003_DB
	x \ ROIN			
889	Time	219.977	120.569	240.061
890	Time	206.995	127.984	229.343
891	Time	202.937	117.602	218.018
892	Time	186.78	117.388	233.849
893	Time	207.924	143.731	230.923
894	Time	218.286	116.973	219.175
895	Time	216.663	134.218	240.887
896	Time	234.666	112.807	230.147
897	Time	210.676	144.348	229.679
898	Time	179.172	144.242	249.072
899	Time	181.246	124.26	238.945
900	ROINr	0	1	2
901	Age	8	8	8
902	Position	39	69	31
903	Size	3.02848e-11	1.0816e-11	2.1632e-11
904	ONOFF	2	0	0
905	TSus			
906	Stim			
907	BaseLine	199.144	136.43	212.611
908	Experimenter	6	6	6
909	AnalysisBitMask	13	13	13
910	nPoints	900	900	900
911	XDelta	0.2	0.2	0.2
912	XOffset	0	0	0
913	XUnit	-1	-1	-1
914	OriginID	6.46762e+24	6.46762e+24	6.46762e+24
915				

Storage model of the database

### **Global and local constants**

The following global constants are defined at the top of the procedure file. Some will need adjustment for different users:

*k\_LabelList* is a list of labels that are stored with the database. The defaults list is (beware the linebreaks):

```
k_LabelList="ROINr;Age;Position;Size;ONOFF;TSus;Stim;BaseLine;Experimenter;AnalysisBitMask;nPoints;XDelta;XOffset;XUnit;OriginID"
```

*k\_NaturalUnits* is a list of SI units that the wave scaling can assume. More can be added at the end, as they are index-coded. However, Igor might not scale them normally. For instance, if the unit is inches ("in"), then thousands of them will become kiloinches ("kin"). Why do you think did we come up with SI units in the first place?

*k\_Experimenter* is a list of people analysing experiments. This helps keeping track of who analyzed an experiment. If this is changed after some databases have been built, add names at the end, as the names are index-coded.

*k\_Template* is the template for names of PopulationWaves that will be added to the database using the [GrepString](#) operation. Associated information is contained in waves that have specific suffixes to their names (see next point).

*k\_Suffixes* specifies the name suffix of waves containing a certain type of information that will be added to the database.

*k\_Autodetect* is a list of labels that will be detected automatically, based on whether waves with the appropriate name+suffix are present

See also: [ExpDB2\\_Extraction](#), [PopulationWave](#), [GrepString](#) and [Regular Expressions](#)

### **BuildDataBaseFromSubFolders** (*[Template]*)

This function screens all subfolder for waves whose name matches *Template* (default: see [Global and local constants](#)) and compiles them into a new database.

### **CombineDataBases** (*ListWave*)

This function combines (proto-)database waves referenced in the reference wave *ListWave* into a single database, combining the labels of all of them, if they are different. See the example of how to make a reference wave.

Example:

```
make/wave ListWave
ListWave = {DataBase1, DataBase2, DataBase3} //no double quotes
CombineDataBases(ListWave)
```

See also: [Wave reference waves](#)

### **DBFooter** (*PopWave*, [*LabelList*, *ResultName*])

This function appends a footer to *PopWave* and thus converts it to a proto-database. The following fields are automatically filled in: ROINr, nPoints, XDelta, XOffset, XUnit, OriginID. The optional parameter *LabelList* specifies the labels that are added (default is the constant *k\_LabelList*). The optional parameter *ResultName* specifies the name of the resulting wave (default: "FO\_" + NameOfWave(*PopWave*)). The function returns the resulting wave reference.

**EDB2\_CP ()**

This function launches the control panel to specify a population wave and waves containing associated information to be compiled into a database. If an existing database is selected, the new information will be appended, otherwise a new proto-database will be created.

**TraceFromDB** (*DataBase*, *index*, [*ResultName*])

This function extracts a single trace from *DataBase* at index *index* and returns it as *ResultName* (the default is the name of the original wave+"\_"+Num2Str(index)).

See also: [PopFromDB](#)

**MakeDataBase** (*wList*, [*LabelList*])

This function makes proto-databases out of the [PopulationWaves](#) referenced in the reference wave *wList* and combines them into a single database. It also checks if waves specified by the local constants *k\_Suffixes* and *k\_AutoDetect* are present and adds their data in the respective fields.

The optional parameter *LabelList* specifies the labels that are added (default is the constant *k\_LabelList*). The resulting database wave is returned by the function. See the example of how to make a reference wave.

Example:

```
make/wave ListWave
ListWave = {PopWave1, PopWave2, PopWave3}      //no double quotes
CombineDataBases(ListWave)
```

See also: [Wave reference waves](#)

**PopFromDB** (*DataBase*, [*ResultName*])

This function removes the footer from *DataBase* and applies the wave scaling, thus making it into a [PopulationWave](#). However, this works only if all the traces in *DataBase* have the same number of points and scaling in the x dimension. The result is returned as *ResultName* (default: NameOfWave(*DataBase*)+"\_Pop")

See also: [TraceFromDB](#)

- **ExpDB2\_Extraction**

This function provides a template to programatically extract information from an [ExpDataBase2](#) database.

Related Topics: [ExpDataBase2](#), [Bitwise and Logical Operators](#)

- **GammaCorrCP**

This control panel allows the change of the gamma value for a grayscale image. It actually creates a lookup table, rather than changing the image values. Note that changing the gamma value of an image constitutes a non-linear adjustment which has to be disclosed in most scientific journals.

## • **HClu**

The functions in HClu are used for the implementation of hierarchical clustering algorithms of time-series data (in the form of a [PopulationWave](#)).

### **NormalizeTraces** (*PopWave, options*)

This function normalizes traces in the PopulationWave *PopWave* from 0 to 1 (*options* bit 0 set) and then divides by its standard deviation (*options* bit 1 set)

See also: [Setting Bit Parameters](#)

### **BinPop** (*PopWave, ResultName, nBins, [smth]*)

This function changes the y data from *PopWave* into *nBins* bins and returns the result as *\$ResultName*. The optional parameter *smth* is the number of smoothing operations to be applied for binomial smoothing (default: 0, ie no smoothing).

See also: [Smooth](#)

### **PopDistances** (*PopWave, options*)

This function calculates a distance matrix of all traces in *PopWave*. The metering method is specified by *options*:

*options* = 0 Euclidean

*options* = 1 Chebychev

*options* = 2 Hamming after binning with 5 bins, smoothing 5

*options* = 3 Chebychev after binning with 5 bins, smoothing 5

*options* = 4 normalized Euclidean

*options* = 5 Pearson distance (Pearson distance = 1 - Pearson's R)

*options* = 6 Manhattan

### **HiClu** (*Sorted, cutoff*)

This function calculates the membership of traces in *sorted* according to *cutoff*. *Sorted* is a wave with three columns, the first one holds the distances between two traces, the second and the third hold the index of the traces that the distance was measured between. *Sorted* is usually the output of the function [DistanceMatrix2Column](#), with the optional parameter *index* = 1 that has been sorted by the function [SortByFirst](#). The resulting wave, DM2C, holds the number of the cluster respective traces, specified by the row (i.e. [p]), belong to. Traces are added by single linkage (a.k.a. nearest neighbor clustering) to clusters, i.e. if the distance to any trace in a cluster is less than *cutoff*.

See also: [HCluCP](#)

### **HiClu2D** (*Sorted*)

This function calculates the nodes where subclusters of traces in *sorted* converge with higher clusters in a dendrogram. *Sorted* is a wave with three columns, the first one holds the distances between two traces, the second and the third hold the index of the traces that the distance was measured between. *Sorted* is usually the output of the function [DistanceMatrix2Column](#), with the optional parameter *index* = 1 that has been sorted by the function [SortByFirst](#). The resulting wave, Clusters2D holds the number of the cluster respective traces, specified by the row (i.e. [p]), belong to. Each column represents one iteration, so that in the first column each trace is a cluster on



its own, and in the last column all traces belong to the same cluster. The wave Cluster\_Dist holds the absolute distances for each column.

See also: [HCluCP](#)

### **SortByFirst** (*DM2C*, *rev*)

This function sorts the three-column wave *DM2C* by the entries in the first column. If *rev* > 0, sorting is done in ascending order, else in descending order. This function returns a wave named NameofWave(*DM2C*)+"\_s".

### **Renumber1D** (*wv*)

This function renumbers the integer numbers in *wv* so that all consecutive numbers are present and none are left out.

- **HiCluCP**

This function launches the control panel for hierarchical clustering. Functions are called in the following order:

(optional) [NormaliseTraces](#)  
[PopDistances](#)  
[DistanceMatrix2Column](#)  
[SortByFirst](#)  
[HiClu](#)

- **IPLPosCP**

This function launches a control panel to access functions to determine the position of regions of interest in a layered structure.

#### Overview

Following steps will be executed:

1. Calculating centers of mass of the regions of interest.
2. Determining the borders of the structure, either by thresholding or free-hand drawing.
3. Interpolating the borders to have 8 times the number of points than the X dimension has pixels and smoothing by applying the robust form of the Loess algorithm.
4. Calculating percentiles between the two borders.
5. Determining the percentile closest to each center of mass.

See also: [CenterOfMass](#), [Interpolate2](#), [IPLPosition](#), [Loess](#)

#### The Panel

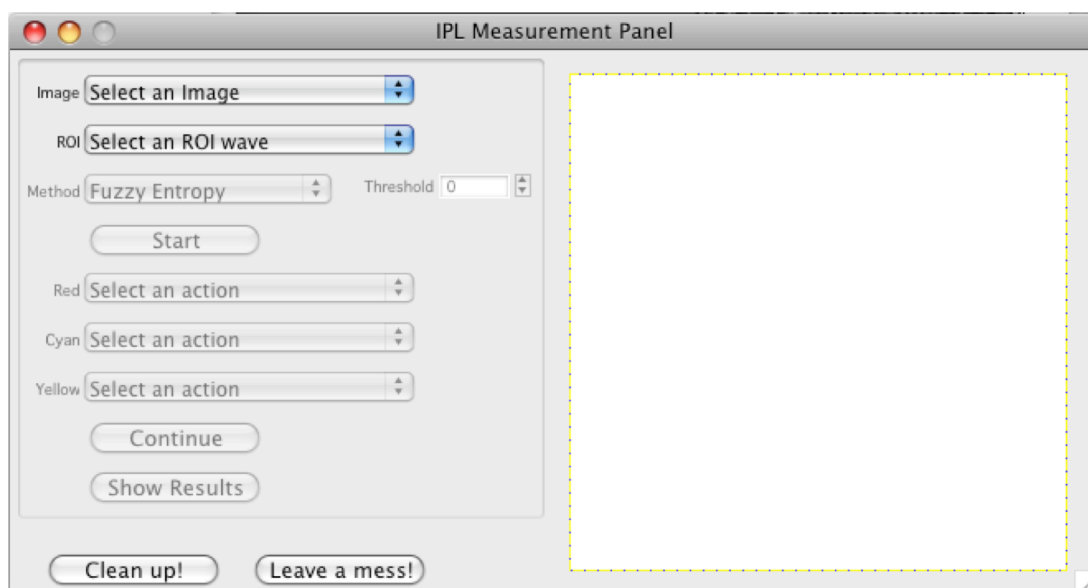
First, a grayscale image of the desired structure and an image specifying the regions of interest have to be specified in the appropriate pull-down menus. The latter has to be a [MultiROI ROI wave](#). Then, a method for thresholding the borders has to be selected. *Iterative* works well, if a bright structure is well separated from the background and no other bright structures are present. If this fails, *Manual Threshold* may still work. Borders can always be specified by *Manual Drawing*, where the user has to draw points along the borders.

When *Start* is clicked, centers of mass for the regions of interest will be calculated and the thresholding applied. If *Manual Drawing* has been selected, then the user will be prompted to draw the borders (see below). The results are then displayed: Three coloured lines: red, cyan and yellow, as well as the centers of mass.

The user has to specify, which of these lines is at the top, i.e. 100% (by definition, this is the photoreceptor side in the inner plexiform layer) and which at the bottom, i.e. 0% (ganglion cell side in the inner plexiform layer). Since automated thresholding sometimes breaks a border, two can be specified for either top or bottom.

Clicking *Continue* will then calculate the percentage for each center of mass and store them in the wave *positions*.

The panel can be closed by clicking the button *Clean Up* or *Leave a Mess!*. The former will keep only the wave positions and kill all intermediates, while the latter will keep all the waves generated in the procedure and only remove global variables associated with the panel.



The IPL Measurement Panel

### Manual Drawing

If *Manual Drawing* has been selected, then clicking *Start* will open a new window in which the background image is shown. By clicking in this window the user can specify points, between which a line will be drawn. When finished, double-click in the window and close it. This procedure is then repeated for the second border. After the second window has been dismissed, the results will be shown in the panel. Note that the lines specified by the user will be interpolated and smoothed.

Related Topics: [CenterOfMass](#), [IPLPosition](#), [MultiROI ROI wave](#)

- **IPLPosition**

These functions are used for determining the positions of regions of interest within a layered structure. They were written to analyse data from the inner plexiform layer

of the retina, but may work for other layered structures as well. These functions are best accessed by the function [IPLPosCP](#) which is in the file IPLPosCP.ipf.

**Note:** Required include files are:

CenterOfMass.ipf

NaNBust.ipf

### **Horizontal**(*Xfunc*, *Yfunc*)

This function returns 1, if *Yfunc* vs *Xfunc* is roughly horizontal, and 0 if it is more vertical.

Horizontality is determined by first dividing the waves by their maximum value, then differentiating them. Then the maxima (maxX and maxY) and minima (minX and minY) of the derivatives are calculated. If the product of the maxX \* minX is larger or equal to the product of maxY \* minY, then 1 returned, else 0.

### **InterpAndLoess** (*XWave*, *YWave*, *interpNumber*, [*SmoothNumber*])

This function interpolates the curve *YWave* vs *XWave* linearly to *interpNumber* of points and then smoothes *YWave* vs *XWave* using locally-weighted regression smoothing with a factor of *SmoothNumber* (0 < SmoothNumber < 1; default = 0.5).

See Also: [Interpolate2](#), [Loess](#)

### **MeasureDistances** (*CoM*, *BottomX*, *BottomY*, *TopX*, *TopY*)

This function calculates the relative distance of centers of mass (stored in *CoM*) from the borders *BottomY* vs *BottomX* and *TopY* vs *TopX*. The top border is by definition 100%, the bottom 0%. The results are stored in the wave Positions.

This is achieved by first calculating percentiles between the two borders and then determining which of the percentile lies closest to each of the points. CoM is a two-dimensional wave, where Y coordinates are stored in CoM[][1] and X coordinates in CoM[][0].

**Note:** Points outside the borders will not be calculated precisely, but are assigned values of -1 or 101. You might want to remove them before doing any statistics.

See Also: [CenterOfMass](#)

### **QuickPA** (*image*, *size*, *method*, [*level*])

This function applies particle analysis using a predetermined set of parameters. *Image* is the wave containing the image to be analysed, *size* is the minimum size in pixels for particles found, *method* specifies the thresholding method (see [ImageThreshold](#) for more details) and the optional parameter *level* is used if manual thresholding (method=0) is selected.

After removing extreme values (i.e. those coinciding with the borders of the image), the three longest contingent borders are returned as the waves

TopY vs TopX,

BottomY vs BottomX, and

ThirdY vs ThirdX

**Note:** While pairs of point match in all of these resulting Y vs X waves, they might

not be ordered in a left-to-right fashion. This depends on the actual symmetry of the object to be detected. The function [reorder](#) has to be used to return them into an ascending order.

The core function of the code is the following:

```
ImageThreshold/I/M=(method)/Q image
ImageAnalyzeParticles /f/E/W/Q/M=0/A=(size) stats, M_ImageThresh
//M_ImageThresh is the result of the ImageThreshold operation
```

See Also: [ImageThreshold](#), [ImageAnalyzeParticles](#)

### **Reorder**(*XWave*,*YWave*)

This operation reorders pairs of points in *YWave* vs *XWave* in ascending order of *XWave*.

### **Thickness**(*Percentiles*)

This operation calculates and prints statistics on the distances between the 0th and 100th percentile in wave *percentiles*. Percentiles are a result of the [MeasureDistances](#) operation. Since the results are calculated using the [WaveStats](#) operation, its automatically created variables can be used.

See also: [WaveStats](#)

## • **LoadScanImage**

The functions in LoadScanImage load gif images, stacks and movies recorded with the ScanImage plugin for Matlab (Pologruto et al. 2003, Journal Biomed Eng Online) and extract the information stored in the header. Files saved with other programs will be loaded, but since the header info is not available, or in a different format, it can't be used to set wave scaling, splitting channels, etc.

**Note:** Two constants are set at the top of the procedure file, which may need to be adjusted between different setups:

*Z\_factor* is the factor by which distances in the z dimension have to be multiplied in order to return actual distances in  $\mu\text{m}$ . This is important only for the correct scaling of image stacks, but not for single images or movies.

*ImageLength* is the side length in  $\mu\text{m}$  of an image taken at zoom 1. This will, among other factors, depend on the objective used. This is important to get the correct X and Y scaling of images.

For time series, the internal clock of the recording computer is assumed to be correct, so no adjustment is being made.

### **LoadScanImage** ( )

This function loads an image or a stack of images and puts them in a 2D or 3D wave that has the same name as the the image file (without the extension). The header will be copied into the wave note. This functions returns the name of the so generated wave as a string.

**Note:** On a Macintosh OSX computer, this operation seems to stall the Igor Pro application with the spinning wait cursor (a.k.a. spinning pizza of death) appearing.

This behaviour is normal (sort of), with the duration of that effect being roughly proportional to the size of the image file being loaded. Other applications work normally in that time.

See also: [ImageLoad](#), [Note](#)

### **LoadMovie** ()

This function loads multiple images into one 3D wave that has the same name as the first image file (without the extension). The header will be copied into the wave note. This function returns the name of the so generated wave as a string. This is useful to load a timelapse movie that has been acquired in ScanImage.

The user is prompted to specify one file. This file is the first in the sequence to be loaded. The last three characters of the filename (not the extension) have to be numbers. This operation will then load all consecutive images in that folder into one 3D wave.

**Note:** On a Macintosh OSX computer, this operation seems to stall the Igor Pro application with the spinning wait cursor (a.k.a. spinning pizza of death) appearing. This behaviour is normal (sort of), with the duration of that effect being roughly proportional to the size of the image file being loaded. Other applications work normally in that time.

See also: [ImageLoad](#), [Note](#)

### **ZoomFromHeader** (*PicWave*)

This function returns the zoom factor from the wave note of *PicWave*.

### **FramesFromHeader** (*PicWave*)

This function returns the number of frames from the wave note of *PicWave*.

### **XRelFromHeader** (*PicWave*)

This function returns the relative X position of the stage from the wave note of *PicWave*.

### **YRelFromHeader** (*PicWave*)

This function returns the relative Y position of the stage from the wave note of *PicWave*.

### **ZRelFromHeader** (*PicWave*)

This function returns the relative Z position of the stage from the wave note of *PicWave*.

### **FilePathFromHeader** (*PicWave*)

This function returns the file path where *PicWave* was loaded from as a string.

### **FileNameFromHeader** (*PicWave*)

This function returns the file name where *PicWave* was loaded from as a string.

**msPerLineFromHeader** (*PicWave*)

This function returns the acquisition time for one line in milliseconds from the wave note of *PicWave*.

**sPerLineFromHeader** (*PicWave*)

This function returns the acquisition time for one line in seconds from the wave note of *PicWave*.

**ExpDateFromHeader** (*PicWave*)

This function returns the date of the experiment (i.e. when the image was acquired) from the wave note of *PicWave* as a string.

**ZSlicesFromHeader** (*PicWave*)

This function returns the number of Z slices from the wave note of *PicWave*. It will be 0, if a movie, rather than a Z stack had been acquired.

**ZStepSizeFromHeader** (*PicWave*)

This function returns the Z step size of slices from the wave note of *PicWave*. This is the number stored in the header, rather than a distance. This number multiplied by the constant *Z\_factor* returns the actual distance (in  $\mu\text{m}$ ).

**nChannelsFromHeader** (*PicWave*)

This function returns the number of channels that *PicWave* was acquired in (currently 1 - 3).

**SplitChannels** (*PicWave*, *nChannels*)

This function splits *PicWave* into *nChannels* stacks, each corresponding to the recording of a separate channel.

**ApplyHeaderInfo** (*Wave3D*)

This function applies X, Y and Z scaling from stored in the wave note to *Wave3D*. If *Wave3D* is a Z stack, then the Z scaling will be a unit of length, if it is a movie, it will be a unit of time, and if it is a single frame, then only X and Y scaling will be set.

**InvertImage** (*Image*)

This function inverts image by calculating  $v\_max - image$  and returning the result as `NameOfWave(image)+"_inv"`.

**LoadMovie** ()

This function lets the user specify a single file on the hard drive. All sequentially numbered .tiff files with the same base name are loaded into an image stack. The images must have the same number of pixels in the x and y dimension. Note that the numbering is taken from the last 3 characters of the filename, e.g., MyFile013.tif would return 13 as the file's number and attempt to load MyFile014.tif next. The function automatically stops if no matching file is found in the same directory. Use this function to load time-lapse data.

See Also: [SetScale](#)

- **ManThresh**

The functions in ManThresh create a panel to threshold a grayscale image. The resulting [MultiROI ROI wave](#) will be saved as MTROIWave.

The thresholding will be performed on the sum of the 2nd derivatives in the X and Y dimension. Optionally, regions of interest smaller than a given size can be excluded.

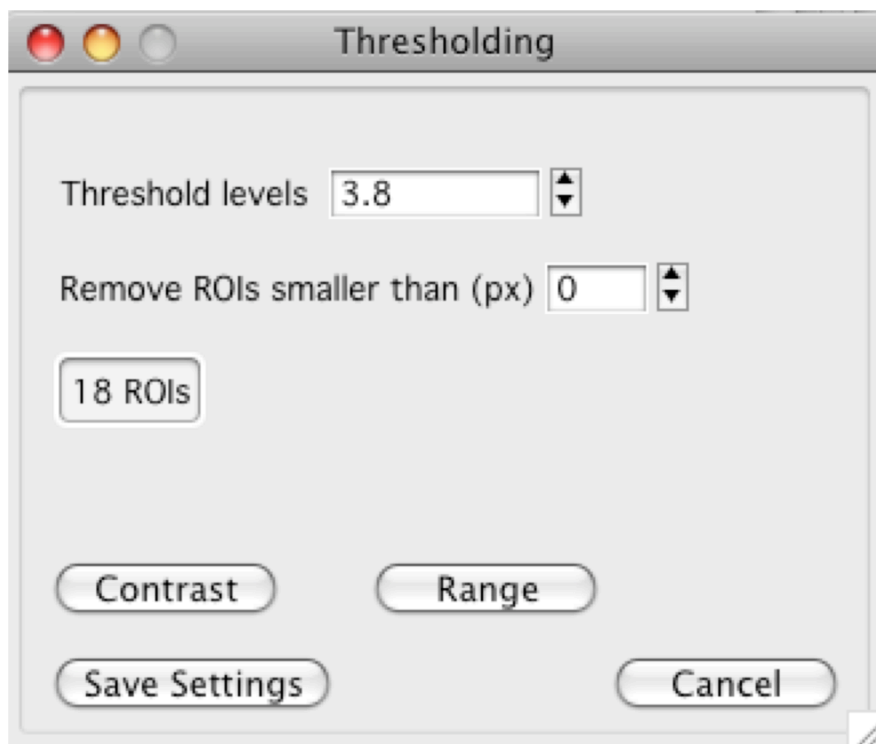
See Also: [dif\\_image](#), [MultiROI ROI wave](#)

**ManThresh** (*picwave*, *startlevels*)

This function opens the thresholding panel. *Picwave* is the grayscale image to be thresholded, *startlevels* is the threshold level to be started with. This number is multiplied by the standard deviation of the 2nd derivative image (see above). Empirically, 3 is a good number to start.

Calling ManThresh opens two new windows: The control panel and a preview window. The user changes the threshold level until they are happy with the preview. By clicking *Save Settings*, the result will be saved as MTROIWave. *Cancel* aborts the operation and saves no wave. *Contrast* and *Range* can be used to change the way *picwave* is displayed in the preview window. A Textbox will show how many regions of interest (ROIs) have been found.

**Note:** The algorithm to remove ROIs smaller than a given size is very slow with resolutions over a few thousand pixels. Using the arrows will make Igor calculate all intermediate steps, so it is better to enter numbers directly.



The ManThresh control panel

## • **MultiROI ROI wave**

The multiROI ROI wave is a concept to encode different regions of interest (ROIs) in a single mask. Inbuilt functions in Igor use a binary ROI in which pixels with a value of 1 lie outside the ROI, those with a value of 0 inside. MultiROI ROI waves encode multiple ROIs using increasing negative numbers (-1 based). The function [MultiROIStack](#) in [Z-Project](#) needs a MultiROI ROI wave passed as a parameter.

### **MultiROI** (*sourcewave, targetwave*)

This operation modifies a binary ROI mask to become a [MultiROI ROI wave](#). *sourcewave* is a binary ROI mask in which pixels with a value of 1 lie outside the ROI, those with a value of 0 inside. The function MultiROI assigns all connected pixels, i.e. pixels that lie in one of the eight surrounding pixels of any pixel with a value of 0, a negative number. It scans first line-by-line, increasing the rows at the end of each line, starting from the origin.

The string *targetwave* will be the name of the so generated MultiROI ROI mask.

**Note:** If two pixels are "connected" only on an edge, they will still be assigned the same ROI number. This behaviour can be changed by commenting out the four appropriate conditional assignments in the code (i.e. where neither x2 nor y2 is 0).

### **MultiROIByLayer** (*sourcewave, targetwave*)

This operation calls [MultiROI](#) on each layer of the 3D wave *sourcewave*. The string *targetwave* will be the name of the so generated MultiROI ROI mask.

### **MultiROIStats** (*image, ROI,[m]* )

This operation calculates statistics of the ROIs defined by *ROI mask* on *image*. Set *m* to 2 in order to calculate the higher statistical moments; The default is 1. The results are stored in the wave ROIStats. Each row corresponds to an ROI, the different statistics are stored in columns as follows:

ROIStats[][0] = v_avg	Average of pixel values.
ROIStats[][1] = v_min	Minimum pixel value.
ROIStats[][2] = v_max	Maximum pixel value.
ROIStats[][3] = V_npnts	Number of points in the ROI.
(m=2 only):	
ROIStats[][4] = v_sdev	Standard deviation of pixel values.
ROIStats[][5] = v_rms	Root mean squared of pixel values
ROIStats[][6] = v_skew	Skewness of pixel values.
ROIStats[][7] = v_kurt	Kurtosis of pixel values.
ROIStats[][8] = v_adev	Average deviation of pixel values.

See Also: [ImageStats](#)

Related Topics: [ImageGenerateROIMask](#), [MultiROI](#), [Z-Project](#)

## • **NaNBust**

The functions in NaNBust replace numbers or NaN entries in waves. They are slightly more comfortable to use than inbuilt functions.



**NaNBust** (*wv*, [*newnum*])

This operation will replace NaN values in the wave *wv* with 0 or the value of the optional parameter *newnum*. The heart of this function is the [MatrixOP](#) function:

```
MatrixOP/o/free NaN_Busted = ReplaceNaNs(wv, newnum)
```

**Replace** (*wv*, *findVal*, *replacementVal*)

This operation will replace values equal to *replace* in the wave *wv* with the value of *with*. The heart of this function is the [MatrixOP](#) function:

```
MatrixOP/o/free w_Replaced = Replace(wv, findVal, replacementVal)
```

**Note:** *replace* can be any number, but not NaN or  $\pm\text{inf}$ . The function NaNBust can be used in the former case.

Related Topics: [MatrixOP](#), [NumType](#), [SelectNumber](#)

- **Normalize**

The functions in Normalize normalize waves and images.

**IACPNormalize** (*sourcewave*, *targetwave*)

This operation normalizes *sourcewave* by dividing it by the average of all points and then subtracting 1. The string *targetwave* is the name of the so calculated wave. Hence, the average of *targetwave* will be 0.

**Note:** This function normalizes to the average of all values in *sourcewave*. Often it is better to normalize to a baseline. This function is useful, however, if a baseline can not be automatically determined.

**Normalize** (*wv*, *from*, *to*, [*name*])

This operation normalizes all values in the wave *wv* so that the values will be spread from *from* to *to*. The optional parameter *name* is the name of the so calculated wave, the default being `nameofwave(wv)+"_nor"`.

This operation first subtracts the smallest value (min) of *wv* from *wv* and then adds *from*. Then, *wv* will be multiplied by *to* and divided by (max - min). Max is the largest value in *wv*.

This is essential when a double precision wave is to be saved as an image, as images can only assume limited values, e.g. from 0 to 255 for 8 bit grayscale, 0 to 65535 for 16 bit grayscale, 0 to 255 in three channels for 24 bit colour, etc.

**NormalizePop** (*sourcewave*, *targetwave*)

This operation normalizes all rows in the populationwave *sourcewave* by dividing each column by its average and then subtracting 1. The string *targetwave* is the name of the so calculated populationwave.

**Note:** Confusingly, columns of a wave will be plotted against the X axis, while rows will be plotted against the Y axis in an image.

**Note:** This function normalizes to the average of a whole column in *sourcewave*. Often it is better to normalize to a baseline. This function is useful, however, if a baseline can not be automatically determined.

See Also: [Populationwave](#)

**Rectify** (*sourcewave*, *targetwave*, [*direction*])

This operation sets all negative values in *sourcewave* to zero. If the optional parameter *direction* is set to a value smaller than zero, all positive values in *sourcewave* will be set to zero instead. The string *targetwave* is the name of the so calculated wave. The heart of this function is the [SelectNumber](#) function:

```
wv_calc = SelectNumber((wv_calc[p][q][r][s])*direction<0,wv_calc[p][q][r][s],0)
```

See Also: [SelectNumber](#)

## • **OneClickSmooth**

The functions in OneClickSmooth provide quick access to Igor's inbuilt image filtering algorithms, as well as filtering 3D data by principal component analysis (PCA).

Filtering by PCA lets the user choose the number of principal components to keep. Therefore, a higher number means less filtering.

**OCS** ()

opens a user interface from which the following parameters can be specified:

Image - The top wave in the top graph will be automatically chosen, but others can be specified from the pull-down menu.

Method - See the [ImageFilter](#), [MatrixFilter](#) and [PCA](#) operations for a detailed description.

Filter Size - In pixels. Odd numbers should be preferred, as these preserve the filter symmetry. This won't have an effect if the method Hybridmedian has been selected. If PCA is selected, this number will specify the numbers of principal components that are not rejected (i.e. kept).

The filtered image will have the same name as the original with "\_fil" appended to its name.

**Note:** For three dimensional waves (i.e. image stacks or movies) the appropriate 3D ( $n \times n \times n$ ) filters will be used, while 2D ( $n \times n$ ) filters will be used for two dimensional data. The method Hybridmedian exists only as a 3D filter, for 2D data the method FindEdges will be used instead. PCA works only on 3D data.

**Filter2** (*image*)

The Filter2 operation works similar to [OCS](#), only that the image to be filtered is passed as the parameter *image*. A popup will prompt the user for Method and Filter Size.

Related Topics: [ImageFilter](#), [MatrixFilter](#), [PCA](#)

## • Populationwave

A populationwave is a 2D wave that contains multiple 1D waves with the same scaling and number of points. The X axis scaling/units of a populationwave is the same as the X axis/units scaling of the 1D waves, while the Y dimension is the index. The data full scale of the populationwave contains the units of the 1D waves.

The functions in Populationwave generate and separate populationwaves.

### AverageWavesFromWindow ()

This operation first stores all traces from the top graph in a populationwave called *WinPop* and then calculates the average, SD and SEM of these traces. The results are stored in the waves *W\_PopAvg*, *W\_PopSD* and *W\_PopSEM*, respectively. Furthermore, the average trace will be displayed as a bold black trace in the top window, and the average trace plus error bars (SD) will be displayed in a new graph.

### PopulationWave (*basename*, *outputname*, *number*, [*offset*])

This function generates a populationwave of *number* 1D waves. All 1D waves must have the same *basename* and end with numbers. The optional parameter *offset* specifies the number of the first 1D wave, the default being zero. The string *outputname* is the name of the so generated populationwave.

### Examples

```
PopulationWave("MyWave_", "MyPopWave", 10)
//Generates a wave named MyPopWave out of the waves MyWave_0, MyWave_1, ...,
MyWave_9

PopulationWave("MyWave_", "AnotherPopWave", 5, offset=10)
//Generates a wave named AnotherPopWave out of the waves MyWave_10, MyWave_11,
..., MyWave_14.
```

### PopStats(*PopWave*)

This operation calculates the average, SD and SEM of all lines in the populationwave *PopWave*. The results are stored in the waves *W\_PopAvg*, *W\_PopSD* and *W\_PopSEM*, respectively.

### PopWaveFromWindow ()

This operation generates a populationwave out of all waves displayed in the top window. The resulting populationwave will be named *WinPop*. This operation works if lines out of a populationwave are displayed in the top window.

**Note:** The number of points and the X and data scaling will be taken from the top wave, as they are assumed to be the same in all waves.

### PopX2Traces (*PopWave*, *basename*)

This operation reverses the operation *PopulationWave*: It generates a 1D wave out of each column of the populationwave *PopWave*. The string *basename* will be the *basename* of the so generated waves.

### PopY2Traces (*PopWave*, *basename*)

This operation is similar to *PopX2Traces*, only that it generates a 1D wave out of each

row of the populationwave *PopWave*. The string *basename* will be the basename of the so generated waves. This is equal to calling [PopX2Traces](#) after [TransposeXY](#).

### **TransposeXY** (*basename*, *outputname*)

This function swaps rows and columns (and the respective scaling) of the wave *basename*. The string *outputname* is the name of the so generated populationwave. Some inbuilt functions in Igor operate along rows, rather than columns.

## • **RegisterStack**

The functions in RegisterStack call the inbuilt operation ImageRegistration with a predefined set of parameters on a stack of images (3D wave).

**Note:** ImageRegistration for stacks has been implemented in Igor Pro version 6.1. For earlier versions, a workaround is implemented that is slower, and probably less efficient.

See also: [ImageRegistration](#)

### **RegisterStack** (*picwave*, [*target*])

This operation registers the imagestack *picwave*. The optional string *target* is the name of the registered stack (default: Nameofwave(*picwave*)+"\_reg").

The core of this operation (for Igor Pro 6.1 or later) is:

```
imageregistration /q /stck /csnr=0 /refm=0 /tstm=0 testwave=regcalcwave,
refwave=ref
```

Regcalcwave is a duplicate of *picwave*, ref is the first frame of *picwave*.

See also: [ImageRegistration](#)

### **Reg2** (*picwave*)

Works similar to [RegisterStack](#), only that *picwave* is overwritten.

### **QuickReg** ()

Calls [Reg2](#) on the top wave of the top window.

## • **ResultsByCoef**

The operations in ResultsByCoef are used to retrieve a subset of data from a wave. They are not called in any automated analysis.

Definition: A coefficients wave as used by operations in this file consists of the elements one is interested in. For instance, if one was interested in retrieving the data from traces #3,4,7 and 8 then one would make a coefficients wave like this:

```
make /o coef={3,4,7,8}
```

**CoMByCoef** (*CoM, coef*)

This operation retrieves paired data from the (2D) wave *CoM* stored at the rows specified in the wave *coef* and stores them in the wave *CbC*. *CoMByCoef* is generally used to retrieve centers of mass but may be applied to other 2D waves that store paired information.

See also: [CenterOfMass](#)

**CoMByLayer** (*Positions, CoM, Layers*)

This operation retrieves paired data from the (2D) wave *CoM* and stores it into the 3D *CoMPoP* wave with the 3rd dimension reflecting the respective layers. The layers are separated by the information in the wave *layers* which stores the center and width ( $=2*\sigma^2$ ) information in the fashion *Layers*={center(0), width(0), center(1), width(1),..., center(n),width(n)} which is compared to the position of each center of mass, as stored in *Positions*. A center of mass is supposed to lie in a particular layer if it is within  $\text{center} \pm \text{width}/2$ .

See also: [IPLPosition](#), [IPLPosCP](#)

**ResultsByCat** (*data, catWave, category*)

This operation retrieves data from the (1D) wave *data* if the variable *category* matches the entry of the wave *catWave* at that index.

**ResultsByCoef** (*data, coef*)

This operation retrieves data from the (1D) wave *data* stored in the points specified in the wave *coef* and stores them in the wave *results*.

Example:

```
make /o data={2,1,3,5,6,8,11,4,7}
make /o coef={0,4,5,7}
resultsbycoef(data,coef)
print results                // prints Results[0]= {2,6,8,4}
```

Related Topics: [CenterOfMass](#), [PopulationWave](#)

- **ROISize**

The operations in *ROISize* determine the sizes of regions of interest (ROIs) in a [multiROI ROI wave](#), remove ROIs below a certain size and renumber them.

**ClearROIMarquee** ()

To use this operation, first display a [multiROI ROI wave](#), then drag a marquee over an area where the ROIs are to be removed. *ClearROIMarquee* then removes all ROIs within a selected marquee and renumbers them.

For quick access, this function should be placed in a menu.

**RemoveROI** (*ROIwave, threshold*)

This operation removes all ROIs in the [multiROI ROI wave ROIwave](#) whose size is smaller than or equal to *threshold*. The result is stored in the wave ROI\_edit.

### **RenumberROI** (*ROIwave*)

This operation renumbers all ROIs in the [multiROI ROI wave ROIwave](#) in order to account for numbers that have been removed.

### **ROISize** (*ROIwave*)

This operation determines the sizes in pixels of regions of interest (ROIs) in the [multiROI ROI wave ROIwave](#) and stores them in the wave Size. More statistics can be calculated using [MultiROIStats](#).

## • **RotateFunction**

The operations in RotateFunction rotate functions (i.e. scaled 1D waves or two 1D waves specifying x and y coordinates waves), images, image stacks or 2-column waves specifying x and y coordinates (such as the output of [CenterOfMass](#)) in the x/y plane.

### **RotateFunction1** (*func, angle, CenterX, CenterY*)

This operation rotates *func* angle degrees around the coordinates specified by *CenterX* and *CenterY*. The x scaling is taken from the scaling of *func*. The output are two waves, *rot\_x* and *rot\_y*, which can be displayed using the following command:

Display *rot\_y* vs *rot\_x*

### **RotateFunction2** (*func, angle, CenterX, CenterY*)

This operation rotates *Yfunc* vs *Xfunc* angle degrees around the coordinates specified by *CenterX* and *CenterY*. The output are two waves, *rot\_x* and *rot\_y*, which can be displayed using the following command:

Display *rot\_y* vs *rot\_x*

### **RotateImage** (*image, angle*)

This operation rotates the image or image stack *image* angle degrees (specified if the origin is in the lower left) around its center. This function preserves the pixel values and looks for the closest matching pixel to put them in, thus no interpolation is performed, which is the main difference to [ImageRotate](#). The output is the wave W\_RotateImage.

### **RotateImage** (*CoM, image, angle*)

This operation rotates *CoM* around the center of *image* by *angle* degrees. *CoM* is a 2-column wave specifying x and y coordinates (such as the output of [CenterOfMass](#)). The output is *CoM\_rot*.

## • **RotateGUI**

RotateGUI() calls a panel to specify parameters for rotating an image or image stack in the x/y plane. Matrix Rotation calls the custom-written function [RotateImage](#), which preserves pixel values, i.e. does not interpolate. Image Rotation, which calls

[ImageRotate](#), however, is much faster.

- **SaveTiff**

This operation saves a 2D or 3D wave as a tiff image.

**Note:** Normalize.ipf is a required include file.

**SaveTiff** (*wv*, [*depth*])

This operation saves the image or image stack *wv* as a tiff file. The optional parameter *depth* specifies the bit depth of the tiff file (default is 16). Only following bit depths are allowed: 1, 8, 16, 24, 32, 40.

The user will be prompted for a location and a file name.

Related Topics: [Normalize](#)

- **Z-Project**

The operations in Z-Project analyse or modify stacks of grayscale images, i.e. movies or volume data.

Definitions:

ROI...region of interest

Image stack...a stack of grayscale images, 4 dimensions (x,y,z, intensity). Stored in a 3D (!) wave

Image...a grayscale image, 3 dimensions (x, y, intensity). Stored in a 2D (!) wave.

Trace...a trace, 2 dimensions (x, y), stored in a 1D (!) wave.

Populationwave...a 2D wave that stores several traces. The trace number is encoded in the y dimension, while the y value of the traces is stored as the intensity.

**Note:** Required include files are:

MultiRoi.ipf

EqualizeScaling.ipf

See also: [EqualizeScaling](#), [PopulationWave](#), [MultiROI ROI wave](#)

**AvgZ** (*picwave*, *outputwave*)

This operation averages all layers of the image stack *picwave* using the inbuilt function:

```
imagetransform averageimage picwave
```

The string *outputwave* will be the name of the so generated image M\_AveImage. The new image will have the same x and y scaling as *picwave*.

See also: [Imagetransform](#)

**StdevZ** (*picwave*, *outputwave*)

This operation averages all layers of the image stack *picwave* and returns the image showing the standard deviation using the inbuilt function:

```
imagetransform averageimage picwave
```

The string *outputwave* will be the name of the so generated image M\_StdvImage. The new image will have the same x and y scaling as *picwave*. The string *outputwave* will be the name of the so generated image stack.

**MaxZ**(*picwave*, *outputwave*)

MaxZ generates a single image out of the stack *picwave* whose pixel intensities represent the maximum values of all layers of *picwave*. The string *outputwave* will be the name of the so generated image. The new image will have the same x and y scaling as *picwave*.

**MinZ** (*picwave*, *outputwave*)

MinZ generates a single image out of the stack *picwave* whose pixel intensities represent the minimum values of all layers of *picwave*. The string *outputwave* will be the name of the so generated image. The new image will have the same x and y scaling as *picwave*.

**RangeZ** (*picwave*, *outputwave*)

RangeZ generates a single image out of the stack *picwave* whose pixel intensities represent the maximum minus minimum values (range) of all layers of *picwave*. The string *outputwave* will be the name of the so generated image. The new image will have the same x and y scaling as *picwave*.

**zStack** (*picwave*, *outputwave*, *roiwave*)

zStack returns a Z-stack of all layers in the image stack *picwave*, averaging all pixels in *picwave* whose value in *roiwave* is 0. Obviously, *picwave* and *roiwave* must have the same number of pixels in the x and y dimension. *roiwave* is two-dimensional. The string *outputwave* will be the name of the so generated trace, whose x scaling will be the same as the z scaling of *picwave*, while its y scaling will not be set.

**MultiROIzstack** (*picwave*, *outputwave*, *roiwave*)

MultiROIzstack returns multiple Z-stacks of all layers in the image stack *picwave*, averaging all pixels in *picwave* that have the same negative value in *roiwave*. The string *outputwave* will be the name of the so generated 2D wave (a [populationwave](#)), whose x scaling will be the same as the z scaling of *picwave*. *roiwave* must be a MultiROI ROI wave, in which increasing negative numbers (-1 based) encode different ROIs.

**SubstBG** (*picwave*, *outputwave*, *roiwave*)

SubstBG returns an image stack with the same scaling as *picwave* that has the average intensity of all pixels with a value of 0 in *roiwave* subtracted from *picwave* on a layer-by-layer basis. Obviously, *picwave* and *roiwave* must have the same number of pixels in the x and y dimension. *roiwave* is two-dimensional. The string *outputwave* will be the name of the so generated image stack.



**NormBG** (*picwave*, *outputwave*, *roiwave*)

NormBG returns an image stack with the same scaling as *picwave*. The result is *picwave* divided by the average intensity of all pixels with a value of 0 in *roiwave* minus 1, on a layer-by-layer basis. Obviously, *picwave* and *roiwave* must have the same number of pixels in the x and y dimension. *roiwave* is two-dimensional. The string *outputwave* will be the name of the so generated image stack.

See also: [PopulationWave](#), [MultiROI ROI wave](#)

**SubstBGPoly** (*picwave*, *outputwave*, *roiwave*, *order*)

This operation calls the function

```
imageremovebackground /r=roiwave /p=(order) frame
```

which removes a general background level, described by a polynomial of a specified order *order*, from the image in *picwave* layer by layer, which is copied into the wave *frame*. The string *outputwave* will be the name of the so generated image stack.

**Note:** This function is experimental and is not called by any automated image analysis procedures.

See also: [ImageRemoveBackground](#)

Related Topics: [EqualizeScaling](#)