1.2.1 image - Tool

Package general
Module images

Operations on images
include "image.g"

Constructors

image	Construct an image tool from an AIPS++ image file
imagecalc	Construct an AIPS++ image via the image calculator
imageconcat	Construct an AIPS++ image by concatenating images
imagefromarray	Construct an AIPS++ image from a Glish array
imagefromascii	Construct an AIPS++ image by conversion from an ascii file
imagefromfits	Construct an AIPS++ image by conversion from a FITS image file
imagefromforeign	Construct an AIPS++ image from a foreign package image file
imagefromimage	Construct a (sub)image from a region of an AIPS++ image
imagefromshape	Construct an empty AIPS++ image from a shape
imagemaketestimage	Construct an AIPS++ image from a test FITS file

Functions

adddegaxes	Add degenerate axes of the specified type to the image
addnoise	Add noise to the image
boundingbox	Get the bounding box of the specified region
brightnessunit	Get the image brightness unit
calc	Image calculator
calcmask	Image mask calculator
close	Close the image tool
continuumsub	Image plane continuum subtraction
convertflux	Convert flux density between peak and integral
convertsm	Convert image for older versions
convolve	Convolve image with a Glish array or another image
convolve2d	Convolve image by a 2D kernel
coordmeasures	Convert from pixel to world coordinate wrapped as Measures
coordsys	Get the Coordinate System of the image
decompose	Separate a complex image into individual components
deconvolvecomponentlist	Deconvolve a componentlist from the restoring beam
delete	Delete the image file associated with this image tool
done	Destroy this image tool
fft	FFT the image
findsources	Find point sources in the sky
fitallprofiles	Fit all 1-dimensional profiles in a region
fitpolynomial	Fit 1-dimensional polynomials to profiles
fitprofile	Fit 1-dimensional profile with functional forms

fitsky	Fit 2-dimensional models to the sky
getchunk	Get the pixel values from a regular region of the image into a Glish array
getregion	Get pixels and mask from a region-of-interest of the image
getslice	Get 1-D slice from the image
hanning	Convolve one axis of image with a Hanning kernel
haslock	Does this image have any locks set ?
histograms	Compute histograms from the image
history	Recover and/or list the history file
id	Return the fundamental identifier of this tool
insert	Insert specified image into this image
isopen	Is this Image `tool` open ?
ispersistent	Is the image persistent ?
lock	Acquire a lock on the image
makecomplex	Make a complex image
maskhandler	Handle pixel masks
maskhandlergui	Handle masks using a GUI interface
maxfit	Find maximum and do parabolic fit in the sky
miscinfo	Get the miscellaneous information record from an image
modify	Modify image with a model
moments	Compute moments from an image
momentsgui	Compute moments from image via custom GUI interface
name	Name of the image file this tool is attached to
open	Open a new image file with this image tool
pixelvalue	Get value of image and mask at specified pixel coordinate
putchunk	Put pixels from a Glish array into a regular region of the image
putregion	Put pixels and mask into a region-of-interest of the image
rebin	rebin the image by the specified factors
regrid	regrid this image to the specified Coordinate System
rename	Rename the image file associated with this image tool
replacemaskedpixels	replace the values of pixels which are masked bad
restoringbeam	Get the restoringbeam
rotate	rotate the direction coordinate axes attached to the image and regrid the i
sepconvolve	Separable convolution
sepconvolvegui	Separable convolution via custom GUI interface
set	Set pixel and/or mask values with a scalar in a region-of-interest of the im
setbrightnessunit	Set the image brightness unit
setcoordsys	Set new Coordinate System
sethistory	Set the history for an image
setmiscinfo	Set the miscellaneous information record for an image
setrestoringbeam	Set the restoringbeam
shape	Length of each axis in the image
statistics	Compute statistics from the image
subimage	Create a (sub)image from a region of the image
summary	Summarize basic information about the image
toascii	Convert the image to an ascii file
tofits	Convert the image to a FITS file

topixel	Convert from world to pixel coordinate
toworld	Convert from pixel to world coordinate
twopointcorrelation	Compute two point correlation function from the image
type	Return the type of this tool
unlock	Release any lock on the image
view	Display the current image with the viewer

Summary
Image tools provides access to AIPS++ images. Currently only single precision floating point AIPS++ images are supported by the Image `tool`. In the future, complex images will also be supported.
Image tools also provide direct (native) access to FITS and Miriad images. You can also convert these foreign formats to AIPS++ format (for optimum processing speed).

Overview of Image `tool` functionality

- **Conversion -** There is functionality to interconvert between AIPS++ images and FITS files. There is also native access to a FITS file:
  - imagefromfits - Convert a FITS image file to an AIPS++ image
  - tofits - convert the image to a FITS file
  - image - native access to a FITS file
  - imagefromascii - Convert an ascii image file to an AIPS++ image
  - imagefromarray - Convert a Glish array into an AIPS++ image
  - imagefromshape - Convert a shape into an AIPS++ image

  Finally, some foreign image formats (e.g. Miriad, GIPSY) can be converted to AIPS++ (if the foreign packages are installed locally) via an intermediate FITS file.
  - imagefromforeign - Currently supported are Miriad and GIPSY files.
- **Analysis -**
  - decompose - separate a complex image into individual components
  - deconvolvecomponentlist - deconvolve a Componentlist from the restoring beam
  - fft - FFT the image
  - findsources - Find strong point sources in sky
  - fitsky - fit 2-dimensional models to the sky (see also the imagefitter `tool`.

  - fitpolynomial - fit 1-d polynomials to profiles and subtract (see also the imageprofilefitter `tool`.
  - fitprofile - fit a 1-d profile with varying combinations of functional forms (see also the imageprofilefitter `tool`.
  - fitallprofile - fit all the 1-d profiles in a region with varying combinations of functional forms and write out the fitted and residual images (see also the imageprofilefitter `tool`.
  - histograms - compute histograms from the image
  - insert - insert specified image into this image
  - maxfit - Find maximum and do simple parabolic fit to sky
  - modify - modify image by a model
  - moments - compute moments from image
  - momentsgui - compute moments from image via custom GUI interface
  - regrid - regrid the image to the specified Coordinate System
  - rotate - rotate the coordinate system and regrid the image to the rotated Coordinate System
  - rebin - rebin an image by the specified binning factors
  - statistics - compute statistics from the image
  - twopointcorrelation - compute two point autocorrelation functions from the image
  - subimage (function) - Create a (sub)image from a region of the image

- **Coordinates -** Manipulation of the coordinate system is handled through

  - coordmeasures - convert from pixel to world coordinate wrapped as Measures
  - coordsys - recover the Coordinate System into a Coordsys `tool`.
  - setcoordsys - set a new Coordinate System
  - topixel - convert from world coordinate to pixel coordinate
  - toworld - convert from pixel coordinate to world coordinate

  The coordsys `tool` provides more extensive coordinate system manipulation.

- **Filtering -** Images may be filtered via

  - convolve - Convolve image with a Glish array or by another image
  - convolve2d - Convolve image by a 2D kernel

  - sepconvolve - Separable convolution
  - sepconvolvegui - Separable convolution via custom GUI interface
  - hanning - Hanning convolution along one axis

  In the future filtering other than convolution will be provided

- **Display** -
  - view - display image with the Viewer
- **Masks** - Masks may be manipulated via
  - calcmask - Image mask calculator
  - maskhandler - handle masks (set, copy, delete, recover names)
  - maskhandlergui - handle masks via a GUI interface
  - replacemaskedpixels - replace the values of pixels which are masked bad
  - set - set pixel and/or mask values with a scalar in a `region-of-interest` of the image
  - summary - lists the mask names
- **Pixel access** - The pixel and mask values for an image may be accessed and calculated with via
  - imagecalc - Create image tool with image calculator
  - image - Create an image tool from an AIPS++ image
  - calc - Image pixel calculator
  - calcmask - Image mask calculator
  - getchunk - get the pixel values from a regular region of the image into a Glish array
  - getregion - get pixels and mask from a `region-of-interest` of the image
  - getslice - get a 1-D slice from the image
  - pixelvalue - get image value for specified pixel
  - putchunk - put pixels from a Glish array into a regular region of the image
  - putregion - put pixels and mask into a `region-of-interest` of the image
  - set - set pixel and/or mask values with a scalar in a `region-of-interest` of the image
- **Inquiry** - Functions to report basic information about the image are

  - boundingbox - find bounding box of a `region-of-interest`.
  - brightnessunit - Get image brightness unit
  - haslock - does this image have a lock set
  - history - recover/list history file
  - id - Return the fundamental identifier of this `tool`
  - ispersistent - is the image persistent (on disk)
  - name - name of the `image file` this tool is attached to
  - restoringbeam - Get restoring beam
  - shape - the length of each axis in the image
  - summary - summarize basic information about the image
  - type - the type of this Image `tool`

- **Utility -** There is wide range of utility services available through the functions
  - adddegaxes - Add degenerate axes
  - addnoise - Add noise to the image
  - brightnessunit - Get image brightness unit
  - close - Close the `image tool` (but don't destroy it)
  - convertflux Convert flux density between peak and integral
  - convertsm Convert the image so that older versions can read it
  - delete - delete the `image file` associated with this Image `tool`
  - done - destroy this `image tool`
  - haslock - does this image have a lock set
  - history - recover/list history file
  - imagefiles - Find the names of all image files in the given directory
  - imagetools - Find the names of all global image `tools`
  - imagedones - Run the done function on the given (or all) global image `tools`
  - is_image - Is this variable an Image `tool`
  - isopen - Is this Image `tool` open ?
  - lock - acquire a lock on the image
  - makecomplex - make a complex image from two real images
  - miscinfo - recover miscellaneous information record
  - open - open a new `image file` with this image tool
  - rename - rename the `image file` associated with this Image `tool`
  - restoringbeam - Get restoring beam

  - setbrightnessunit - Set image brightness unit
  - sethistory - set the history file
  - setmiscinfo - set the miscellaneous information record
  - setrestoringbeam - Set new restoring beam
  - unlock - release lock on this `image file`

- **Reshaping** - Images can be reshaped via
  - imagefromimage (constructor) - Create a (sub)image from a region of an AIPS++ image
  - subimage (function) - Create a (sub)image from a region of the image
  - insert - insert specified image into this image
  - imageconcat - Concatenate AIPS++ images

- **Tests and demos** -
  - imagetest - Run test suite for Image `tool`
  - imagedemo - Demonstrate the use of an image

General
We refer to an AIPS++ `image file` when we are referring to the actual data stored on disk. The name that you give an AIPS++ `image file` is actually the name of a directory containing a collection of AIPS++ tables which together constitute the `image file`. But you only need to refer to the directory name and you can think of it as one *logical* file.
Whenever we use the word "image", we are just using it in a generic sense. AIPS++ images are manipulated with an Image tool. An `image tool` is associated with, or bound to, the actual `image file` via a `constructor`. Note that some `image tools` don't have a disk file associated with them. These are called "virtual" images and are discussed below
When an image is stored on disk, it can, in principle, be stored in a variety of ways. For example, the image could be stored row by row; this is the way that most older generation packages store images. It makes for very fast row by row access, but very slow in other directions (e.g. extract all the profiles along the third axis of an image). An AIPS++ `image file` is stored with what is called tiling. This means that small multi-dimensional chunks (a tile) are stored sequentially. It means that row by row access is a little slower, but access speed is essentially the same in all directions. This in turn means that you don't need to (and can't !) reorder images.
Here are some simple examples using image tools.

Example

```
- include 'image.g'                    # Load image tool functionality
- include 'logger.g'                   # Load logger tool functionality
- dl.gui()                             # Display logger GUI
- im := imagemaketestimage('zz')       # Make test image; writes disk file called 'zz'
- im.summary()                         # Summarize (to logger)
- im.view()                            # Display image
- im.statistics()        # Evaluate statistics over entire image
#
- box := drm.box([10,10], [50,50])     # Make a pixel box region with regionmanager
- im2 := im.subimage('zz2', box)       # Make a subimage  called 'zz2'
- im2.statistics()                     # Evaluate statistics
- im2.fft (amp='zz2.amp', phase='zz2.phase')  # FFT subimage and store amp and phase
#
- im.done()                            # Release tool resources (destroys it) - disk f
- im2.done()
```

Foreign Images

The Image `tool` also provides you with native access to some foreign image formats. Presently, these are FITS (Floar, Double, Short and Long are supported) and Miriad. This means that you don't have to convert the file to native AIPS++ format in order to access the image. For example:

Example

```
- include 'image.g'
- ima := image('im.app')        # Access aips++ image
- imf := image('im.fits')       # Access FITS image
- imm := image('im.mir')        # Access Miriad image
-
- ims := imagefromimage(infile='im.fits', region=drm.quarter())
```

Each of these Image tools has access to all the same `tool functions`.
Where ever you see an argument in an Image `tool` constructor which is an input image disk file, that disk file can be an AIPS++, FITS, or Miriad image file.
There are some performance penalties that you should be aware of. Firstly, because AIPS++ images are tiled (see above) you get the same access speed regardless of how you access the image. FITS and Miriad images are not tiled. This means that the performance for these Image tools will be poorer for certain operations. For example, extracting a profile along the third axis of an image, or re-ordering an image with the display library.
Secondly, for FITS images, masked values are indicated via "magic value". This means that the mask is worked out on the fly every time you access the image.

If you find performance is not good enough or you want a writable image, then use appropriate constructor (imagefromfits or imagefromforeign) to convert to a native AIPS++ image.

Virtual Images
We also have Image `tools` that are not associated one-to-one with disk files; these are called "virtual" images (see also the article in the AugustNewsLetter). For example, with the image calculator, imagecalc, one can create an expression which may contain many images. You can write the result of the expression out to a disk `image file`, but if you wish, you can also just maintain the expression, evaluating it each time it is needed - nothing is ever written out to disk in this case. There are other Image constructors and functions like this (the documentation for each one explains what it does). The rules are:

- If you specify the `outfile` argument, then the image is always written to the specified disk `image file`.
- If you leave the `outfile` argument unset, then if possible, a virtual image will be created. Sometimes this virtual image will be an expression as in the example above (i.e. it references other images) or a temporary image in memory, or a temporary image on disk. (the summary function will list for you the type of image you have). When you destroy that Image tool, the virtual image will be destroyed as well.
- If you leave `outfile` unset, and the function or constructor cannot make a virtual image, it will create a disk file for you with a name of its choice (usually input plus function name).
- You can always write a virtual image to disk with the subimage `tool function`.

Coordinate Systems
An image contains a Coordinate System. A Coordsys `tool` is used to manipulate the Coordinate System. An Image `tool` allows you to recover the Coordinate System into a Coordsys `tool` through the coordsys) function. You can set a new Coordinate System with the setcoordsys function.
You can do some direct coordinate conversion via the Image `tool` functions toworld, topixel, and coordmeasures. The actual work is done by a Coordsys `tool`, for which these Image `tool` functions are just wrappers.

Lattice Expression Language (LEL)
LEL allows you to manipulate expressions involving images. For example, add this image to that image, or multiply the miniumum value of that image by the square root of this image. The LEL syntax is quite rich and is described in detail in note 223.
LEL is accessed via the imagecalc constructor and the calc `tool` function. Here are some examples.

Example

```
- include 'image.g'
- im := imagemaketestimage()       # Make virtual test image
- im.calc('$im + min($im)')        # Make the minimum value zero
-
- im2 := imagemaketestimage('zz')   # Make nonvirtual test image
- im2.calc('zz + min(zz)')          # Make the minimum value zero
-
- im.done()
- im2.done()
```

In the first example, the Image `tool` **im** is associated with a virtual image. We use the $ syntax in the expression accessing this image via its `tool`. In the second example, the Image `tool` **im2** is associated with the non-virtual disk file *zz*. We could still have used the $ syntax if we wanted, but we show here how you can also use the `image file` name in the expression.
Note that for image file names with special characters in them (like for example), you should (double) escape those characters or put the file name in double quotes. E.g.

Example

```
- im1 := imagecalc(pixels='test\\-im')     # Note double escape required
- im2 := imagecalc(pixels='"test-im"')
```

Region-of-interest
A `region-of-interest` or simply, region, designates which pixels of the image you are interested in for some (generally) astrophysical reason. This complements the `pixel mask` (see below) which specifies which pixels are good or bad (for statistical reasons). `Regions-of-interest` are generated and manipulated with the Regionmanager tool.
Briefly, a `region-of-interest` may be either a simple shape such as a multi-dimensional box, or a 2-D polygon, or some compound combination of `regions-of-interest`. For example, a 2-D polygon defined in the X and Y axes extended along the Z axis, or perhaps a union or intersection of regions. See the Regionmanager documentation for more details on regions.
Regions are always supplied to constructors and `tool functions` via the `region` argument.

Pixel mask
A `pixel mask` specifies which pixels are to be considered good (value `T`) or bad (value `F`). For example, you may have imported a FITS file which has blanked pixels in it. These will be converted into `pixel mask` elements whose

values are bad (F). Or you may have made an error analysis of an image and computed via a statistical test that certain pixels should be masked out for future analysis.
If there is no pixel mask, all pixels are considered good (if you retrieve the pixel mask when there is none, you will get an all good mask). Pixels for which the pixel mask value is bad are not used in computations (e.g. in the calculation of statistics, moments or convolution).
The image may contain zero, one, or more pixel masks. However, only one mask will be designated as the default mask. This is the pixel mask that is actually applied to the data. You can also indicate that none of the pixel masks are the default, so that effectively an all good pixel mask is applied.
The function summary includes in its summary of the image the names of the masks (the first listed, if not in square brackets, is the default).
Pixel masks are handled with the function maskhandler. This allows you to find the names of pixel masks, delete them, copy them, nominate the default and so on. It is not used to change the value of pixel masks.
The functions with which you can change pixel mask values are putregion (put Boolean array), calcmask (put result of Boolean LEL expression), and set (put scalar Boolean).

The argument 'mask'
There is an argument, mask, which can be supplied to many constructors and functions. It is supplied with either a mask region-of-interest (generated via the function wmask) or a LEL Boolean expression string (the same string you would have supplied to the above Regionmanager function). Generally, one just supplies the expression string.
The LEL expression is simply used to generate a pixel mask which is then applied in addition to any default pixel mask in the image (a logical OR).
For example

```
- im := imagemaketestimage('zz')
- im.statistics(mask='zz>0')        # Only evaluate for positive values
- im.calcmask (mask='(2*$im) > 0')  # Create a new mask which is T (good) when twice the
                                    # image values are positive, else F
```

The mask expression must in general conform (shape and coordinates) with the image (i.e. that associated with the Image tool). You can use the $ substitution syntax for Image tools (see example above) which is necessary for virtual images and useful otherwise.
When mask is used with function calcmask, a persistent pixel mask is created and stored with the image. With all other functions and constructors, the mask argument operates as a transient (or On-The-Fly [OTF]) pixel mask. It can be very handy for analysing or displaying images with different masking criteria.
Often I will refer to the "total input mask". This is the combination (logical OR) of the default pixel mask (if any) and the OTF mask (if any).

In the following example we open a Rotation Measure image. We then evaluate statistics and display it where only those pixels whose error in the Rotation Measure (`image file` *rmerr*) is less than the specified value are shown; the others are masked. The nice thing is you can experiment with different `pixel masks` until you are satisfied, whereupon you might then make the `pixel mask` persistent with the calcmask function.

Example

```
- include 'image.g'
- im := image('rm')
- im.statistics(mask='rmerr<10')
- im.view(mask='rmerr<20')
- im.calcmask (mask='rmerr<20')     # Make persistent mask
```

Finally, a subtlety that is worth explaining.

Example

```
- im.statistics(mask='zz>0')                  # Mask of zz ignored
- im.statistics(mask='mask(zz) && zz>0')      # Mask of zz used
```

In the first example, any default mask associated with the image *zz* is ignored. Only the pixel values are looked at. In the second example, the mask of *zz* is also taken into account via the LEL `mask` function. That is, the transient output mask is T (good) only when the mask of *zz* is T and the expression `zz>0` is T.

Example

A useful part of LEL to use with the `mask` argument is the `indexin` function. This enables the user to specify a mask based upon selected pixel coordinates (or indices) rather than image values. For example

```
- im := imagefromshape(shape=[20])
- local pp,mm
- im.getregion (pp,mm, mask='indexin(1, [5:10, 15, 19:20])')
- print mm
[F F F F T T T T T T F F F F T F F F T T]
```

You can see the mask is good (T) for the specified indices along the specified axis. You can also pass in a premade variable for the specification if you like, viz.

```
- im := imagefromshape(shape=[20])
- axis := 1
- sel := [5:10, 15, 19:20]
- local pp,mm
- im.getregion (pp,mm, mask='indexin($axis, $sel)')
- print mm
[F F F F T T T T T T F F F F T F F F T T]
```

This capability is useful for fitting functions. See the example in the fitpolynomial function.

Pixel masks and Regions

Some comment about the combination of `pixel masks` and `regions-of-interest` is useful here. See the Regionmanager tool for basic information about `regions-of-interest` first.

Regions are provided to Image `tool functions` via the standard `region` constructor and function argument.

Consider a simple polygonal region. This `region-of-interest` is defined by a bounding box, the polygonal vertices, and a mask called a `region mask`. The `region mask` specifies whether a pixel within the bounding box is inside or outside the polygon. For a simple box `region-of-interest`, there is obviously no need for a `region mask`.

Now imagine that you wish to recover the `pixel mask` of an image from a polygonal `region-of-interest`. The mask is returned to you in regular Boolean array. Thus, the shape of the returned mask array reflects the bounding-box of the polygonal region. If the actual `pixel mask` that you apply is all good, then the retrieved mask would be good inside of the polygonal region and bad outside of it. If the actual `pixel mask` had some bad values in it as well, the retrieved mask would be bad outside of the polygonal region. Inside the polygonal region it would be bad if the `pixel mask` was bad. More simply put, the mask that you recover is just a logical "and" of the `pixel mask` and the `region mask`; if the `pixel mask` is T *and* the `region mask` is T then the retrieved mask is T (good), else it is F (bad).

Finally, note that if you use the `region` and `mask` (the OTF mask) arguments together then they operate as follows. The shape of the Boolean expression provided by `mask` must be the same shape as the image to which it is being applied. The `region` is applied equally to the image and the `mask` expression. For example

Example

```
include 'image.g'
- rm1 := image('rm')
- rm2 := image('rmerr')
- rm1.shape(); rm2.shape()
```

```
[128 128]
[128 128]
- r := drm.box([10,10], [50,50])
- rm1.statistics(region=r, mask='rmerr<10')       # region applied to 'rmerr' and 'rm'
```

Error Handling
All the constructors and functions for Image `tools` return a variable by value. Because of the weak typing of *Glish*, this variable may take on different forms depending upon the outcome. For example, a constructor such as `im := image('hcn.xyv')` normally returns an `image tool`. However, if the operation failed (e.g. the `image file` *hcn.xyv*) does not exist, them the return variable will be what is called a *Glish* fail. You can check any *Glish* variable with the `is_fail(variable)` function; it returns T or F. With an Image tool, all errors will cause the returned variable, whether it comes from a constructor or a method, to be a fail. You will not get a false (`F`) return unless this is a legal return value for the function. This is so that you always know what to check the return variables value for if you are doing careful error checking in a script. The documentation for each constructor and function always tells you what the return variable type could potentially be.

Example

```
- include 'image.g'
- im := image('rm')
- if (is_fail(im)) {
  note (im::message, origin='myscript', priority='SEVERE')
  return;
}
```

The scoping operator :: enables to extract out a diagnostic message from the variable `im` when it is a fail.

Vector Inputs
Many of the functions of an Image `tool` take vectors (numeric or strings) as their arguments. We take advantage of the weak-typing of Glish so that whenever you see a vector argument, you can assume that

- For numeric vectors you can enter as an actual numeric vector (e.g. [1.2,2.5,3]), a vector of strings (e.g. "1.2 2.5 3") or even a string with white space and/or comma delimiters (e.g. '1.2 2.5 3').
- For string vectors you can enter as an actual vector of strings (e.g. "1.2 2.5 3") or a string with white space or comma delimiters (e.g. '1.2 2.5 3').

Events
An `image tool` may emit five events. These are generated by the view display function. See it for details of the event values.

- A `region` event. This signifies that a `region-of-interest` has been interactively generated via the Viewer.
- A `position` event. This signifies that a position has been interactively generated via the Viewer.
- A `statistics` event. This signifies that statistics have been interactively generated via the Viewer.
- a `breakfromviewer` event. This signifies that the user has pressed the 'break' button on the Viewer display panel.
- a `viewerdone` event. This signifies that the private Viewer inside the Image `tool` has been destroyed.

Short-hand names for functions
Many of the function names are rather long to type (e.g. boundingbox, replacemaskedpixels). This doesn't matter with a GUI interface, but can be annoying if you are writing scripts or using Glish interactively. For such long-winded function names, a short-hand synonym is defined and given at the beginning of the documentation for each function.

Miscellaneous
At present only real-valued (single precision) images are supported. Complex images will be supported in the future.
Besides the functions available as `tool functions`, you may of course add functionality through *Glish* programming. *Glish* computing speed will be adequate for many "chunk-by-chunk" algorithms. Beware putting large images into an array – this can use up a lot of memory!
Some of the `tool functions` in this module can be run asynchronously (usually things that might take a long time). This is controlled by the argument `async` which defaults to `!dowait` (i.e. the opposite to the value of this *Glish* Boolean variable). By default when you start *Glish* `dowait=F`. You can enter `dowait := T` if you wish the default to be that these tool functions run synchronously. If you don't want to type this in every time you start *Glish* then put this statement in your *.glishrc* resource file that lives in your home directory.
Note that if the `tool function` is run asynchronously, you have to recover the result in a more indirect fashion as follows in the following example.

```
- jobnum := im.statistics(async=T)
- statsout := defaultservers.result(jobnum)
```

image - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an image tool from an AIPS++ image file

Synopsis
image(infile)

Arguments

infile	in	Input image file name Allowed:String

Returns
`image tool` or fail

Description

This is the most commonly used constructor. It constructs (creates) a `image tool` associated with disk `image file`.
The disk image file may be a native AIPS++, FITS (Float, Double, Short, Long are supported) , or Miriad `image file`. Look here for more information on foreign images.

Example

```
- im := image('myimage')          # Access native aips++ image
- im.view()
-
- im2 := image('hcn.fits')        # Access FITS image
- im2.view()
```

---

imagecalc - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an AIPS++ image via the image calculator

Synopsis
imagecalc(outfile, pixels, overwrite)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
pixels	in	LEL expression Allowed:String
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
Image tool

Description

This constructor is used to evaluate a mathematical expression involving existing `image files`, assigning the result to the newly created output image. The syntax of the expression supplied via the `pixels` argument (in what is called the Lattice Expression Language, or LEL) is explained in detail in note 223.
Any image files embedded in the expression may be native AIPS++ or FITS (but not yet Miriad) `image files`. Look here for more information on foreign images.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` actually references all of the input files in the expression. So if you deleted any of the input image disk files, it would render this `tool` useless. When you destroy this `tool` (with the done function), the reference connections are broken.
If you make a reference `tool`, the expression is re-evaluated every time you use it. So it takes no extra storage, but takes longer to access the pixel values (they must be recomputed).

Example

```
im:= imagecalc(outfile='cF', pixels='aF + min(float(pi()), mean(bF))')
```

Suppose *aF* and *bF* are `image files` with single precision data. They do not need to have conforming shapes and coordinates, because only the mean of *bF* is used. Note that `pi` is explicitly converted to single precision, because `pi()` results in a Double. If that was not done, the expression result would be a Double with the effect that all data of *aF* had to be converted to Double. So this expression first computes a scalar, the minimum of π and the mean of the `image file` *bF*. That scalar is then added to the value of each pixel in the `image file` *aF*. The resultant is put in the `image file` called *cF* to which access is provided via the `image tool` called **im**.
If the expression is masked, that mask will be copied to the new image.

Example

```
- im:=imagecalc('cF', 'min(aF, (min(aF)+max(aF))/2)')
```

This example shows that there are 2 `min` functions. One with a single argument returning the minimum value of that image. The other with 2 arguments returning an image containing *aF* data clipped at the value of the 2nd argument.

Example

```
- im := imagecalc('eF', 'sin(aD)+(aF*2)+min(bF)+real(aC)')
```

This shows a mixed type expression. The real part of the Complex image *aC* is used in an expression that otherwise uses Float type. The result of the expression is written out to the disk file called *eF*.

Example

```
- im := imagecalc(pixels='sin(aD)+(aF*2)+min(bF)+real(aC)')
```

This shows a mixed type expression. The real part of the Complex image *aC* is used in an expression that otherwise uses Float type. The result of the expression is not written out to disk in this case. It is evaluated each time it is needed.

---

imageconcat - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an AIPS++ image by concatenating images

Synopsis
imageconcat(outfile, infiles, axis, relax, tempclose, overwrite)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
infiles	in	List of input AIPS++ image files to concatenate; wild cards accepted Allowed:String or vector of strings
axis	in	Concatenation pixel axis Allowed:positive integer Default: Spectral axis if there is one
relax	in	Relax constraints that axis coordinate descriptors match Allowed:Bool Default: F
tempclose	in	Keep all lattices closed until needed Allowed:Bool Default: T
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
Image tool

Description

This constructor is used to concatenate two or more input AIPS++ images into one output image. For example, if you have two image cubes which are contiguous along one axis (say a spectral axis) and you would like to glue them together along this axis, then this constructor is the appropriate thing to use. If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` actually references all of the input files. So if you deleted any of the input image disk files, it would render this `tool` useless. When you destroy this tool (with the done function) the reference connections are broken.
The input and output images must be of the same dimension. Therefore, if you wish to concatenate 2-D images into a 3-D image, the 2-D images must

have a third axis (of length unity) so that the output image coordinates are known along the concatenation axis.

The input images are concatenated in the order in which they are listed. Therefore, the Coordinate System of the output image is given by that of the first input image. The input images are checked to see that they are contiguous along the concatenation axis and an error is generated if they are not. In addition, the coordinate descriptors (e.g. reference pixel, reference value etc) for the non-concatenation axes must be the same or an error will result.

The input disk image files may be in native AIPS++, FITS, or Miriad formats. Look here for more information on foreign images.

The contiguous criterion and coordinate descriptor equality criteria can be relaxed by setting `relax=T` whereupon only warnings will be issued. Dimension and shape must still be the same though. When the concatenation axis is not contiguous (but still monotonically increasing or decreasing) and `relax=T`, a tabular coordinate will be used to correctly describe the axis. But be aware that it means adjacent pixels are not regularly spaced. However, functions like toworld and topixel will correctly interconvert world and pixel coordinates.

In giving the input image names, the `infiles` argument can be a single string if you wild card it with standard shell symbols. For example, `infiles='cena_???.*'`, where the "?" represents one character and "*" any number of characters.

Otherwise, you must input a vector of strings such as `infiles="cena1 cena2 cena3"`. An input such as `infiles='files1,file2'` will be interpreted as one string naming one file and you will get an error. The reason for this is that although the latter could be parsed to extract two file names by recognizing comma delimiters, it is not possible because an expression such as `infiles='cena.{a,b}'` (meaning files of name "cena.a" and "cena.b") would confuse such parsing (you would get two files of name *cena.*{*a* and *b*}.

You can look at the Coordinate System of the output image with summary to make sure it's correct.

The argument `tempclose` is, by default, True. This means that all internal reference copies of the input images are kept closed until they are needed. Then they are opened temporarily and then closed again. This enables you to effectively concatenate as many images as you like without encountering any operating system open file number limits. However, it comes at some performance loss, because opening and closing all those files takes time. If you are concatenating a smallish number of files, you might use `tempclose=F`. This will leave all internal reference copies permanently open, but performance, if you don't hit the file limit, will be better.

Example

```
- im := imageconcat(outfile='bigimage', infiles=''im1 im2 im3'', axis=3, relax=T, tempc
```

The `image files` *im1*, *im2*, and *im3* are concatenated along the third axis to form the output `image file` *bigfile*. Only warnings are issued if axis descriptors mismatch. Access to the output image is provided via the `tool` called **im**.

Example

```
- im := imageconcat(infiles='$ADATA/cena*.1344')
```

All images whose file names begin with *cena* and terminate with *.1344* that reside in the directory given by the translation of the environment variable `$ADATA` are concatenated along the spectral axis if there is one. The image must be contiguous along the spectral axis and all image coordinate descriptors must match. If there is no spectral axis an error will result. Because an output file name is not given, the Image tool just references the input files, it does not copy them.

---

imagefromarray - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an AIPS++ image from a Glish array

Synopsis
imagefromarray(outfile, pixels, csys, linear, overwrite, log)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
pixels	in	Numeric Glish array
csys	in	Coordinate System Allowed:Coordsys tool Default: unset
linear	in	Make a linear Coordinate System if csys not given Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
log	in	Write image creation messages to logger Allowed:T or F Default: T

Returns
Image tool or fail

Description

This constructor converts a *Glish* array of any size into an AIPS++ `image file`.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
At present, no matter what type the `pixels` array is, a real-valued image will be written (the input pixels will be converted to Float with the *Glish* as_float function). In the future, Complex images will be supported.
The Coordinate System, provided as a Coordsys `tool`, is optional. If you provide it, it must be dimensionally consistent with the pixels array you give (see also coordsys).
If you don't provide the Coordinate System (unset), a default Coordinate System is made for you. If `linear=F` (the default) then it is a standard RA/DEC/Stokes/Spectral Coordinate System depending exactly upon the shape of the `pixels` array (Stokes axis must be no longer than 4 pixels and you may find the spectral axis coming out before the Stokes axis if say, `shape=[64,64,32,4]`). Extra dimensions are given linear coordinates. If `linear=T` then you get a linear Coordinate System.

Example

```
- im := imagefromarray(outfile='test.data', pixels=array(0, 64, 64, 4, 128))
- cs := im.coordsys(axes=[1,2])
- im.done()
- im2 := imagefromarray(pixels=array(1.0, 32, 64), csys=cs)
- cs.done()
```

The first example creates a zero-filled `image file` named *test.data* which is of shape [128,256]. If you examine the header with `im.summary()` you will see the default RA/DEC/Stokes/Frequency coordinate information. In the second example, a Coordinate System describing the first two axes of the image *test.data* is created and used to create a 2D image temporary image.

---

imagefromascii - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an AIPS++ image by conversion from an ascii file

Synopsis
imagefromascii(outfile, infile, shape, sep, csys, linear, overwrite)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
infile	in	Input ascii disk file name Allowed:String
shape	in	Shape of image Allowed:Vector of integers
sep	in	Separator in ascii file Allowed:String Default: ' '
csys	in	Coordinate System Allowed:Coordsys tool Default: unset
linear	in	Make a linear Coordinate System if csys not given Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
Image tool or fail

Description

This constructor is used to convert an ascii disk file to an AIPS++ `image file`. If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
You must specify the shape of the image. The image must be stored in the ascii file, one row at a time. This row must be of the correct length and there must be the correct number of rows. For example, let us say the shape of your image is `[nx,ny,nz] = [3,4,2]`, then the image should be stored as

```
1 1 1                   # row 1; y = 1, z = 1
2 2 2                   # row 2; y = 2, z = 1
3 3 3                   # row 3; y = 3, z = 1
4 4 4                   # row 4; y = 4, z = 1
1.5 1.5 1.5             # row 5; y = 1, z = 2
2.5 2.5 2.5             # row 6; y = 2, z = 2
3.5 3.5 3.5             # row 7; y = 3, z = 2
4.5 4.5 4.5             # row 8; y = 4, z = 2
```

The Coordinate System, provided as a Coordsys `tool`, is optional. If you provide it, it must be dimensionally consistent with the pixels array you give (see also coordsys).
If you don't provide the Coordinate System (unset), a default Coordinate System is made for you. If `linear=F` (the default) then it is a standard RA/DEC/Stokes/Spectral Coordinate System depending exactly upon the shape of the `pixels` array (Stokes axis must be no longer than 4 pixels and you may find the spectral axis coming out before the Stokes axis if say, `shape=[64,64,32,4]`). Extra dimensions are given linear coordinates. If `linear=T` then you get a linear Coordinate System.

Example

```
- im := imagefromascii('myimage.app', 'myimage.ascii', shape=[128,128])
```

---

imagefromfits - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an AIPS++ image by conversion from a FITS image file

Synopsis
imagefromfits(outfile, infile, whichrep, whichhdu, zeroblanks, overwrite, old)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
infile	in	Input FITS disk file name Allowed:String
whichrep	in	If this FITS file contains multiple coordinate representations, which one should we read Allowed:positive integer Default: 1
whichhdu	in	If this FITS file contains multiple images, which one should we read Allowed:positive integer Default: 1
zeroblanks	in	If there are blanked pixels, set them to zero instead of NaN Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
old	in	Use old non-WCS based FITS parser? Allowed:T or F Default: F

Returns
Image tool or fail

Description

This constructor is used to convert a FITS disk image file (Float, Double, Short, Long are supported) to an AIPS++ `image file`. If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
This constructor reads from the FITS primary array (when the image is at the beginning of the FITS file; `whichhdu=1`), or an image extension (when the image is elsewhere in the FITS file, `whichhdu` $>$ `1`).
By default, any blanked pixels will be converted to a mask value which is false, and a pixel value that is NaN. If you set `zeroblanks=T` then the pixel value will be zero rather than NaN. The mask will still be set to false. See the function replacemaskedpixels if you need to replace masked pixel values after you have created the image.

Example

```
- im := imagefromfits('data/myimage', 'fits/MYIMAGE.FITS')
- mi := im.miscinfo()
print 'fields=', field_names(mi)
```

The FITS image is converted to an AIPS++ `image file` and access is provided via the `image tool` called **im**. Any FITS header keywords which were not recognized or used are put in the miscellaneous information bucket accessible with the miscinfo function. In the example we list the names of the fields in this record.

---

imagefromforeign - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an AIPS++ image from a foreign package image file

Synopsis
imagefromforeign(outfile, infile, format, overwrite)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
infile	in	Input foreign image disk file name Allowed:String
format	in	Format type Allowed:String from 'miriad', 'gipsy' Default: 'miriad'
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
Image tool or fail

Description

This constructor is used to convert a foreign package image file to an AIPS++ image. This is done via an intermediary FITS file which is created by the foreign package. In order for this constructor to work, you must have the foreign package installed; it is not part of AIPS++. Currently supported are Miriad and Gipsy files.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.

Example

```
- im := imagefromforeign('im.app', 'im.miriad', format='miriad')
```

---

imagefromimage - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct a (sub)image from a region of an AIPS++ image

Synopsis
imagefromimage(outfile, infile, region, mask, dropdeg, overwrite)

Arguments

outfile	in	Output sub-image file name Allowed:String Default: unset
infile	in	Input image file name Allowed:String
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
dropdeg	in	Drop degenerate axes Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
Image tool or fail

Description

This constructor applies a `region-of-interest` to an `image file`, creates a new `image file` containing the (sub)image, and associates a new `image tool` with it.
The input image file may be in native AIPS++, FITS, or Miriad format. Look here for more information on foreign images.
If `outfile` is given, the (sub)image is written to the specified disk file.
If `outfile` is unset, the Image `tool` actually references the input image file. So if you deleted the input image disk file, it would render this `tool` useless. When you destroy this `tool` (with the done function) the reference connection is broken.
Sometimes it is useful to drop axes of length one (degenerate axes). Use the `dropdeg` argument if you want to do this.
The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too.
See also the subimage function.

Example

```
- im := imagefromimage(outfile='image.small', infile='image.large', region=drm.quarter())
```

The specified `region-of-interest` takes a quarter by area of the first two axes of the image, and all pixels of other axes.

---

imagefromshape - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an empty AIPS++ image from a shape

Synopsis
imagefromshape(outfile, shape, csys, linear, overwrite, log)

Arguments

outfile	in	Name of output image file Default: unset
shape	in	Shape of image
csys	in	Coordinate System Allowed:Coordsys tool Default: unset
linear	in	Make a linear Coordinate System if csys not given ? Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
log	in	Write image creation messages to logger Allowed:T or F Default: T

Returns
Image tool

Description

This constructor creates an AIPS++ `image file` with the specified shape. All the pixel values in the image are set to 0.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary

image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
The Coordinate System, provided as a Coordsys `tool`, is optional. If you provide it, it must be dimensionally consistent with the pixels array you give (see also coordsys).
If you don't provide the Coordinate System, a default Coordinate System is made for you. If `linear=F` (the default) then it is a standard RA/DEC/Stokes/Spectral Coordinate System depending exactly upon the shape (Stokes axis must be no longer than 4 pixels and you may find the spectral axis coming out before the Stokes axis if say, `shape=[64,64,32,4]`). Extra dimensions are given linear coordinates. If `linear=T` then you get a linear Coordinate System.

Example

```
- im := imagefromshape('test.data', [64,64,128])
- cs := im.coordsys(axes=[1,3])
- im2 := imagefromshape(shape=[10, 20], csys=cs)
```

The first example creates a zero-filled `image file` named *test.data* of shape [256,512]. If you examine the header with `im.summary()` you will see the RA/DEC/Spectral coordinate information. In the second example, a Coordinate System describing the first and third axes of the image *test.data* is created and used to create a 2D temporary image.

---

imagemaketestimage - Constructor

Package	**general**
Module	**images**
Tool	**image**

Construct an AIPS++ image from a test FITS file

Synopsis
imagemaketestimage(outfile)

Arguments

outfile	in	Output image file name Allowed:String Default: unset

Returns
Image tool

Description
This function converts a FITS file resident in the AIPS++ system into an AIPS++ image.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.

Example

```
- im := imagemaketestimage()     # make virtual image
- im.view()                      # view it
```

---

image.adddegaxes - Function

Package	**general**
Module	**images**
Tool	**image**

Add degenerate axes of the specified type to the image

Synopsis
adddegaxes(outfile, direction, spectral, stokes, linear, tabular, overwrite)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
direction	in	Add direction axes ? Allowed:T or F Default: F
spectral	in	Add spectral axis ? Allowed:T or F Default: F
stokes	in	Add Stokes axis ? Allowed:Stokes Default: ”
linear	in	Add linear axis ? Allowed:T or F Default: F
tabular	in	Add tabular axis ? Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
Image tool or fail

Description

This function (short-hand name `ada`) adds degenerate axes (i.e. axes of length 1) of the specified type. Sometimes this can be useful although you will generally need to modify the coordinate system of the added axis to give it the coordinate you want (do this with the Coordsys `tool`).
You specify which type of axes you want to add. You can't add an axis type that already exists in the image. For the Stokes axis, the allowed value (a string such as I, Q, XX, RR) can be found in the Coordsys constructor.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.

Example

```
- im1 := imagemaketestimage()
```

```
- im1.shape()
[113 76]
- im1.coordsys().axiscoordinatetypes()
Direction Direction
-
- im2 := im1.adddegaxes(spectral=T)
- im2.shape()
[113 76 1]
- im2.coordsys().axiscoordinatetypes()
Direction Direction Spectral
-
- im3 := im2.adddegaxes(stokes='Q')
- im3.shape()
[113 76 1 1]
- im3.coordsys().axiscoordinatetypes()
Direction Direction Spectral Stokes
```

In this example, all the images are virtual (temporary images).

---

image.addnoise - Function

Package	**general**
Module	**images**
Tool	**image**

Add noise to the image

Synopsis
addnoise(type, pars, region, zero)

Arguments

type	in	Type of distribution Allowed:String - see below Default: 'normal'
pars	in	Parameters of distribution Allowed:Numeric vector Default: [0,1]
region	in	Region of interest Allowed:Region tool Default: Whole image
zero	in	Zero image first ? Allowed:T or F Default: F

Returns
T or fail

Description

This function adds noise to the image. You may zero the image first before the noise is added if you wish.
The noise can be drawn from one of many distributions, which are also available in the randomnumbers `tool`.
For each distribution, you must supply the type via the `type` argument (minimum match is active) and parameters via the `pars` argument. Each distribution type is described in detail in the randomnumbers `tool`. Briefly:

- binomial – the binomial distribution models successfully drawing items from a pool. Specify two parameters, n and p, respectively. n is the number of items in the pool, and p, is the probability of each item being successfully drawn. It is required that $n > 0$ and $0 \leq p \leq 1$.
- discreteuniform – models a uniform random variable over the closed interval. Specify two parameters, the low and high values, respectively. The low parameter is the lowest possible return value and the high parameter is the highest. It is required that $low < high$.
- erlang – Specify two parameters, the mean and variance, respectively. It is required that the mean is non-zero and the variance is positive.
- geometric – Specify one parameter, the probability. It is required that $0 \leq mean < 1$.
- hypergeometric – Specify two parameters, the mean and the variance. It is required that the variance is positive and that the mean is non-zero and not bigger than the square-root of the variance.

- normal – Specify two parameters, the mean and the variance. It is required that the variance is positive.
- lognormal – Specify two parameters, the mean and the variance. It is required that the supplied variance is positive and that the mean is non-zero.
- negativeexponential – Supply one parameter, the mean.
- poisson – Specify one parameter, the mean. It is required that the mean is non-negative.
- uniform – Model a uniform random variable over a closed interval. Specify two parameters, the low and high values. The low parameter is the lowest possible return value and the high parameter can never be returned. It is required that $low < high$.
- weibull – Specify two parameters, alpha and beta. It is required that the alpha parameter is not zero.

Example

```
- im1 := imagemaketestimage()
- im1.addnoise(type='normal', pars=[0.5, 1], zero=T)
- im1.statistics()
NORMAL: Starting image::statistics
NORMAL: Selected bounding box [1, 1] to [113, 76]
NORMAL: Creating new statistics storage lattice of shape [9]
NORMAL:
NORMAL: Number points =   8.588000e+03       Sum      =   4.258988e+03
NORMAL: Flux density  =   1.972103e+02 Jy
NORMAL: Mean          =   4.959231e-01
NORMAL: Variance      =   9.983749e-02       Sigma    =   3.159707e-01
NORMAL: Rms           =   5.880184e-01
NORMAL:
NORMAL: Minimum value  -6.234988e-01 at [4, 39] (00:00:27.733, +00.00.12.000)
NORMAL: Maximum value   1.664114e+00 at [105, 11] (23:59:33.867, -00.05.24.000)
NORMAL:
NORMAL: Finished image::statistics
       0.07 real        0.03 user           0 system
T
```

A test image is created, zeroed, and noise of mean 0.5 and variance 1 from a normal distribution added.

---

image.convolve - Function

Package **general**
Module **images**
Tool **image**

Convolve image with a Glish array or another image

Synopsis
convolve(outfile, kernel, scale, region, mask, overwrite, async)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
kernel	in	Convolution kernel - array or image filename Allowed:Numeric Glish array or string
scale	in	Scale factor Allowed:Float Default: Autoscale
region	in	Region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
Image tool or fail

Description

This function does Fourier-based convolution of an `image file` by the given kernel.

If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
The kernel is provided as a multi-dimensional *Glish* array or as the filename of a disk-based `image file`. The provided kernel can have less dimensions than the image being convolved. In this case, it will be padded with degenerate axes. An error will result if the kernel has more dimensions than the image.
No additional scaling of the kernel is provided yet.
The scaling of the output image is determined by the argument `scale`. If you leave it unset, then the kernel is normalized to unit sum. If you do not leave `scale` unset, then the convolution kernel will be scaled (multipled) by this value.
Masked pixels will be assigned the value 0.0 before convolution.
The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too.
See also the other convolution functions convolve2d, sepconvolve and hanning.

Example

```
- dowait := T
- r1 := drm.quarter()
- im := image('pks1333-33.6cm')
- kernel := mykernel(...)                          # Your code to generate the kernel
- im2 := im.convolve(kernel=kernel, region=r1)     # Makes temporary image
-
- im3 := im.convolve (kernel=im.name(F))           # Convolve by self
```

---

image.boundingbox - Function

Package	**general**
Module	**images**
Tool	**image**

Get the bounding box of the specified region

Synopsis
boundingbox(region)

Arguments

region	in	The region of interest Allowed:Region tool Default: Whole image

Returns
Glish Record or fail

Description

This function (short-hand name `bb`) finds the bounding box of a region of interest when it is applied to a particular image. It is returned in a *Glish* record which has fields `'blc'`, `'trc'`, `'inc'`, `'bbShape'`, `'regionShape'`, `'imageShape'`, `'blcf'` and `'trcf'` containing the bottom-left corner, the top-right corner (in absolute image pixel coordinates), the increment (stride) of the region, the shape of the boundingbox, the shape of the region, the shape of the image, the blc in formatted absolute world coordinates and the trc in formatted absolute world coordinates, respectively.
Note that the shape of the bounding box will be different from the shape of the region if a non-unit stride (increment) is involved (see the example below).

Example

```
- im := image('hcn')                          # Create image tool
- x := dq.quantity([3,6,9,6,5,5,3],'pix')       # X vector, in absolute pixels
- y := dq.quantity([3,4,7,9,7,5,5],'pix')       # Y vector, in absolute pixels
- r1 := drm.wpoly(x,y,csys=im.coordsys())     # Create polygonal world region
- bb := im.boundingbox(r1)                    # Find bounding box
- print bb
[blc=[3 3] , trc=[9 9] , inc=[1 1], bbShape=[7 7] , regionShape=[7 7], imageShape=[155 178] ]
```

Example

```
- im := image('z1.app')
- b := drm.box([10,10],[20,20],[2,3])
- im.boundingbox(b)
[blc=[10 10] , trc=[20 19] , inc=[2 3] , bbShape=[11 10] , regionShape=[6 4] , imageShape=[113 76
```

In this example we see the difference between bbShape and regionShape because of the increment (stride). See also that the trc is modified by the increment.

image.brightnessunit - Function

Package	**general**
Module	**images**
Tool	**image**

Get the image brightness unit

Synopsis
brightnessunit()

Returns
String or fail

Description

This function (short-hand name `bu`) gets the image brightness unit.

Example

```
- im := image('hcn')
- im.brightnessunit()
Jy/beam
```

---

image.calc - Function

Package	**general**
Module	**images**
Tool	**image**

Image calculator

Synopsis
calc(pixels)

Arguments

pixels	in	LEL expression Allowed:String

Returns
T or fail

Description

This function is used to evaluate a mathematical expression involving AIPS++ images, assigning the result to the current (already existing) image. It complements the imagecalc constructor. See note 223 which describes the the syntax and functionality in detail.
If the expression, supplied via the `pixels` argument, is not a scalar, the shapes and coordinates of the image and expression must conform.
If the image (that associated with the tool) has a `pixel mask`, then only pixels for which that mask is good will be changed. See the function maskhandler for managing image `pixel masks`.
See the related functions set and putregion.

Example

```
- im.calc('min(aF, (min(aF)+max(aF))/2)')
- im.calc('1.0')
```

The first example shows that there are 2 `min` functions. One with a single argument returning the minimum value of that image. The other with 2 arguments returning an image containing "aF" data clipped at the value of the 2nd argument. The second example sets all good pixels to unity.

Example

```
- im.calc('sin(aD)+(aF*2)+min(bF)+real(aC)')
```

This shows a mixed type expression. The real part of the complex image "aC" is used in an expression that otherwise uses Float type.

---

image.calcmask - Function

Package **general**
Module **images**
Tool **image**

Image mask calculator

Synopsis
calcmask(mask, name, default)

Arguments

mask	in	Boolean LEL expression Allowed:Boolean LEL expression or mask region
name	in	Mask name Allowed:String Default: Auto new name
default	in	Make specified mask the default mask ? Allowed:Bool Default: T

Returns
T or fail

Description

This function is used to create a new `pixel mask` via a Boolean LEL expression. This gives you much more scope than the simple set and putregion functions.
See note 223 which describes the the syntax and functionality of LEL in detail.
If the expression is not a scalar, the shapes and coordinates of the image and expression must conform. If the expression is a scalar then the entire `pixel mask` will be set to that value.
By default (argument `name`) the name of a new `pixel mask` is made up for you. However, you specify a `pixel mask` name (use function summary or maskhandler to see the mask names) then it is used. If the `pixel mask` already exists, it is overwritten.
You can specify whether the new `pixel mask` should be the default mask or not. By default, it is made the default `pixel mask` !

Example

```
- im.calcmask(T)                  # Automatically converted to a string
- im.calcmask('zz>0')             # Mask of zz ignored
- im.calcmask('mask(zz) && zz>0') # Mask of zz included
- im.calcmask('$im>min($im)')     # Use tool names
- im.calcmask('zz>min(zz:nomask)')  # Mask of zz not used in scalar function
```

The first example is the equivalent of `im.set(pixelmask=T)`. It sets the entire mask to True.
The second example creates a new `pixel mask` which is True when the pixel values in image *zz* are greater than 0.
Now for some subtlety. Read carefully ! Any LEL expression can be thought of as having a value and a mask. Usually the value is Float and the mask Boolean. In this case, because the expression is Boolean itself, the value is also Boolean. The expression mask would just be the mask of *zz*. Now what `calcmask` does is create a mask from the expression value (which is Boolean) and discards the expression mask. Therefore, the resulting mask is independent of any mask that *zz* might have.
If you wish the mask of the expression be honoured as well, then you can do as in the third example. It says the output `pixel mask` will be True if the current `pixel mask` of *zz* is True and the expression value is True.
The fourth example is like the second, except that we use the pixel values associated with the **im** Image tool itself rather than using its disk name (the Image tool might be virtual so there might not be a disk file). Note one further subtlety here. When the scalar function `min` evaluates a value from `$im`, the default mask of `$im` *will* be used. All the scalar functions look at the mask. If you didn't want the mask to be used you can use the special `:nomask` syntax shown in the final example.

---

image.close - Function

Package	**general**
Module	**images**
Tool	**image**

Close the image tool

Synopsis
close()

Returns
T or fail

Description

This function closes the `image tool`. This means that it detaches the tool from its `image file` (flushing all the changes first). The `image tool` is "null" after this change (it is not destroyed) and calling any `tool function` other than open will result in an error.

Example

```
- im := image('myimage')  # Starts server and creates Image tool
- im.close()              # Closes Image tool
- im.summary()            # 'im' is now null, this results in an error.
- im.open('myotherimage') # 'im' is valid again
```

---

image.continuumsub - Function

Package	**general**
Module	**images**
Tool	**image**

Image plane continuum subtraction

Synopsis
continuumsub(outline, outcont, region, channels, pol, fitorder, overwrite)

Arguments

outline	in	Output line image filename Allowed:String Default: unset
outcont	in	Output continuum image filename Allowed:String Default: 'continuumsub.im'
region	in	Region over which continuum subtraction is desired Allowed:Region Default: whole image
channels	in	Channels to use for continuum estimation Allowed:Vector¡Int¿ Default: All
pol	in	Polarization to process Allowed:String from "I Q U V XX RR" etc. Default: unset
fitorder	in	Polynomial order for continuum estimation Allowed:Int Default: 0
overwrite	in	Auto-overwrite output files if they exist? Allowed:Boolean Default: F

Returns
Image tool containing the continuum-subtracted result.

Description

This function packages the relevant image tool functionality for simple specification and application of image plane continuum subtraction. All that is required of the input image is that it have a non-degenerate Spectral axis. The user specifies region, the region of the input image over which continuum subtraction is desired (otherwise the whole image will be treated); channels, the subset of channels on the spectral axis to use in the continuum estimation, specified as a vector; pol, the desired single polarization to treat (otherwise all pols, if a polarization axis exists, will be treated); fitorder, the polynomial order to use in the estimation. Optionally, output line and continuum images may be written by specifying outline and outcont, respectively. If outline is not specified, a virtual image tool is all that is produced. If outcont is not specified, the output continuum image will be written in 'continuumsub.im'. In principle, the region may be specified interactively, e.g., using the viewer. At the moment, such regions are typically degenerate on the spectral axis, and so must be manually extended along the this axis, or continuumsub will complain about insufficient spectral channels in the region of interest. Interface improvements in the management of regions currently under consideration will improve the mechanism for doing this.

Example

```
- im1 := image('test1.im');
- im1csub:=im.continuumsub(region=drm.quarter(),channels=[3:8,54:60],fitorder=0);
```

In this first example, the continuum emission in each pixel of the inner quarter of the input image is estimated by averaging (fitorder=0) channels 3-8 and 54-60. im1csub is an image tool containing the result (only the inner quarter of the original image).

```
- im2 := image('test2.im');
- im2csub:=im.continuumsub(channels=[3:8,54:60],pol='I',fitorder=2);
```

In this second example, the Stokes I continuum emission in each pixel of the whole input image is estimated with a 2nd-order polynomial fit to channels 3-8 and 54-60. The output image tool (im2csub) contains only Stokes I, even if the original image had other stokes planes.

---

image.convertflux - Function

Package	**general**
Module	**images**
Tool	**image**

Convert flux density between peak and integral

Synopsis
convertflux(value, major, minor, type, topeak)

Arguments

value	in	Flux density to convert Allowed: Quantity
major	in	Major axis of component Allowed: Quantity
minor	in	Minor axis of component Allowed: Quantity
type	in	Type of component Allowed: String from Gaussian, Disk Default: Gaussian
topeak	in	Convert to peak or integral flux desnity Allowed: Bool Default: T

Returns
Quantity or fail

Description

This function interconverts between peak and integral flux density for a Gaussian or Disk component. The image must hold a restoring beam.

Example

```
- im := image('in.im');
- p1 := dq.quantity('1mJy/beam')
- i1 := im.convertflux(p1, major='30arcsec', minor='10arcsec', topeak=F);
- p2 := im.convertflux(i1, major='30arcsec', minor='10arcsec', topeak=T)
```

```
- print 'peak, integral, peak = ', p1, i1, p2
peak, integral, peak =  [value=1, unit=mJy/beam] [value=0.000163961301, unit=Jy] [valu
```

---

image.convertsm - Function

Package	**general**
Module	**images**
Tool	**image**

Convert image for older versions

Synopsis
convertsm(outfile, overwrite)

Arguments

outfile	in	Output file name Allowed:String Default: unset
overwrite	in	Overwrite output file if already exists ? Allowed:Bool Default: T

Returns
T or fail

Description

`image files` created by some versions of AIPS++ since release 1.3 cannot be be read by versions 1.3 and earlier. The period for which development versions of AIPS++ were creating `image files` which were unreadable by earlier versions was a 2-3 weeks so it is very unlikely you will ecnounter this problem. This function converts the image disk file (actually just the Storage Manager of the logtable) so that it can be read by all versions.
If the `outfile` is unset, the conversion is done in-situ. Otherwise you should specify the name of a new disk file.

Example

```
- im := image('in.im');
- im.convertsm()                        # in-situ
- im.convertsm('out.im', overwrite=T)  # new output file
```

---

image.convolve2d - Function

Package	**general**
Module	**images**
Tool	**image**

Convolve image by a 2D kernel

Synopsis

convolve2d(outfile, axes, type, major, minor, pa, scale, region, mask, overwrite, async)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
axes	in	Axes to convolve Allowed:Vector (length 2) of integers Default: [1,2]
type	in	Type of convolution kernel Allowed:String from 'gaussian' Default: 'gaussian'
major	in	Major axis Allowed:Quantity, string, numeric
minor	in	Minor axis Allowed:Quantity, string, numeric
pa	in	Position Angle Allowed:Quantity, string, numeric Default: 0deg
scale	in	Scale factor Allowed:Float Default: Autoscale
region	in	Region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
Image tool or fail

Description

This function (short-hand name `c2d`) does Fourier-based convolution of an `image file` by the given 2D kernel.
If `outfile` is unset, the image is written to the specified disk file. If `outfile` is not given, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.

You specify which 2 pixel axes of the image you wish to convolve via the `axes` argument.
You specify the type of convolution kernel with `type` (minimum match is active); currently only `'gaussian'` is available. With time others will be implemented.
You specify the parameters of the convolution kernel via the arguments `major`, `minor`, and `pa`. These arguments can be specified in one of three ways

- Quantity - for example `major=dq.quantity(1, 'arcsec')` Note that you can use pixel units, viz. `major=dq.quantity(1, 'pix')`, see below.
- String - for example `minor='1km'` (i.e. one that the Quanta Constructor accepts).
- Numeric - for example `major=10`. In this case, the units of `major` and `minor` are assumed to be in pixels. Using pixel units allows you to convolve unlike axes. For the position angle, units of degrees are assumed.

The interpretation of `major` and `minor` depends upon the kernel type.

- Gaussian - `major` and `minor` are the Full Width at Half Maxiumum (FWHM) of the major and minor axes of the Gaussian.

The position angle is measured North through East when you convolve a plane holding a celestial coordinate (the usual astronomical convention). For other axis/coordinate combinations, a positive position angle is measured from +x to +y in the absolute pixel coordinate frame (x is the first axis that you specify with argument `axes`).
The scaling of the output image is determined by the argument `scale`. If you leave it unset, then autoscaling will be invoked.
If you are not convolving the sky, then autoscaling means that the convolution kernel will be normalized to have unit volume so as to conserve flux.
If you are convolving the sky, then there are two cases for which autoscaling is useful.
Firstly, if the input image units are Jy/pixel, then the output image will have units of Jy/beam and be appropriately scaled. In addition, the restoring beam of the output image will be the same as the convolution kernel.
Secondly,if the input image units are Jy/beam, then the output image will also have units of Jy/beam and be appropriately scaled. In addition, the restoring beam of the output image will be the convolution of the input image restoring beam and the convolution kernel.
If you do not leave `scale` unset, then the convolution kernel will be scaled by this value (it has peak unity before application of this scale factor).
Masked pixels will be assigned the value 0.0 before convolution. The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too.

See also the other convolution functions convolve, hanning, and sepconvolve.

Example

```
dowait := T
im := image('xfy')                                              # RA/Freq/DEC
im2 := im.convolve2d(outfile='xyf.con', axes=[1,3], type='gauss',
                     major='20arcsec', minor='10arcsec', pa=45);
#
im := image('xyf')                                              # RA/DEC/Freq
im2 := im.convolve2d(outfile='xyf.con', axes=[1,3], type='gauss',
                     major='20pix', minor='10pix', pa=45);
```

In the second example we must use pixel units because axes 1 and 3 are unlike.

---

image.coordsys - Function

Package	**general**
Module	**images**
Tool	**image**

Get the Coordinate System of the image

Synopsis
coordsys(axes)

Arguments

axes	in	Axes to which the Coordinate System pertains Allowed: Vector of integers Default: All axes

Returns
coordsys tool or fail

Description

This function stores the Coordinate System of an image in a **Coordsys** tool.

By default, the Coordinate System describes all of the axes in the image. If you desire, you can select a subset of the axes, thus reducing the dimensionality of the Coordinate System. This may be useful if you are supplying a Coordinate System to the constructors imagefromarray or imagefromshape.

Example

```
- im := image('hcn')
- im.summary()
Image name       : hcn
Date observation : 2002/12/31/15:15:36
Image mask       : None
Image units      : JY/BEAM

Direction system : J2000
Frequency system : LSRK
Velocity  system : RADIO
Rest Frequency   : 1.414e+09 Hz

Name             Proj Shape Tile   Coord value at pixel    Coord incr Units
-------------------------------------------------------------------------
Right Ascension   SIN   155   31  17:42:29.303    90.00 -1.000000e+00 arcsec
Declination       SIN   178   30 -28.59.18.600    90.00  1.000000e+00 arcsec
Frequency               256   32  1.413350e+09     1.00  1.968717e+04 Hz
Velocity                          1.378053e+02     1.00 -4.174021e+00 km/s
-
-
- cs := im.coordsys([1,2])
- imshape := im.shape()[[1,2]]
- im2 := imagefromshape(outfile='test', shape=imshape, csys=cs)
- im2.summary()
Image name       : test
Image mask       : None
Direction system : J2000
Rest Frequency   : Absent

Name             Proj Shape Tile   Coord value at pixel    Coord incr Units
-------------------------------------------------------------------------
Right Ascension   SIN   155  155  17:42:29.303    90.00 -1.000000e+00 arcsec
Declination       SIN   178  178 -28.59.18.600    90.00  1.000000e+00 arcsec
```

In this example, we create a Coordinate System pertaining to the first two axes of the image and create a new (empty) 2D image with it with the

`imagefromshape` constructor.

---

image.coordmeasures - Function

Package	**general**
Module	**images**
Tool	**image**

Convert from pixel to world coordinate wrapped as Measures

Synopsis
coordmeasures(pixel, intensity, direction, frequency, velocity)

Arguments

pixel	in	Absolute pixel coordinate Allowed:Numeric vector Default: Reference pixel
intensity	out	Returned intensity quantity for pixel Allowed:Quantity
direction	out	Returned direction measure for pixel Allowed:Direction measure
frequency	out	Returned frequency measure for pixel Allowed:Frequency measure
velocity	out	Returned radial velocity measure for pixel Allowed:Radio definition RadialVelocity measure

Returns
Glish record or fail

Description

You can use this function to get the world coordinates for a specified absolute pixel coordinate in the image. You specify a pixel coordinate (1-rel) for each axis in the image.
If you supply fewer pixel values then there are axes in the image, your value will be padded out with the reference pixel for the missing axes. Excess values will be ignored.
The world coordinate is returned as a record of measures. This function is just a wrapper for toworld (invoked with argument `format='m'`. Please see its

documentation for discussion about the formatting and meaning of the measures.
This Image `tool` function adds two additional fields to the return record. The `mask` field contains the value of the image `pixel mask` at the specified position. It is either T (pixel is good) or F (pixel is masked as bad or the specified position was off the image).
The `intensity` field contains the value of the image (at the nearest pixel to that given) and its units. This is actually stored as a Quantity. This field does not exist if the specified pixel coordinate is off the image.

Example

```
- im := image('myimage')
- meas := im.coordmeasures(im.shape()/2)
- field_names(meas)                          # Names of measures fields in record
intensity mask direction
- print meas.intensity
[value=-0.1478955, unit=JY/BEAM]
- print meas.direction
[type=direction, refer=J2000, m1=[value=-0.505949617, unit=rad],
- m0=[unit=rad, value=-1.64713591]]
-
- dir := meas.direction                      # Get direction coordinate
-
- dm.doframe(dm.observatory('ATCA'))         # Set location on earth
T
- dm.doframe(dm.epoch('utc', '16jun1999/12:30:20'))   # Set epoch
T
- azel := dm.measure(dir, 'azel')            # Convert to azimuth/elevation
-
- print 'az,el=', dq.angle(azel.m0), dq.angle(azel.m1)     # Format nicely
az,el= +092.54.56.777 +068.59.59.576
```

In this example we first find the world coordinates of the centre of the image. Then we use the default Measures `tool` **dm** to convert the `direction coordinate` field from J2000 to an azimuth and elevation at a particular location at a particular time.

---

image.decompose - Function

Package	**general**
Module	**images**
Tool	**image**

Separate a complex image into individual components

Synopsis
decompose(region, mask, simple, threshold, ncontour, minrange, naxis, fit, maxrms, maxretry, maxiter, convcriteria)

Arguments

region	in	Allowed: Region tool Default: unset
mask	in	Allowed: Boolean LEL expression or mask region Default: unset
simple	in	Skip contour deblending and scan for local maxima Allowed: T or F Default: F
threshold	in	Value of minimum contour Allowed: Positive Float Default: unset
ncontour	in	Number of contours to use in deblending Allowed: Int of 2 or greater Default: 11
minrange	in	Minimum number of closed contours in a component Allowed: Positive Integer Default: 1
naxis	in	Max number of perpendicular steps between contiguous pixels Allowed: 1, 2, or 3 Default: 2
fit	in	Fit to the components after deblending? Allowed: T or F Default: T
maxrms	in	Maximum RMS of fit residuals to not retry fit Allowed: Positive Float Default: unset
maxretry	in	Maximum number of times to retry the fit Allowed: Non-negative Integer Default: unset
maxiter	in	Maximum number of iterations allowed in a single fit Allowed: Positive Integer Default: 256
convcriteria	in	Criterion to establish convergence Allowed: Positive Float Default: 0.0001

Returns
Nothing useful yet - component list is printed to screen

Description
The function decompose is an image decomposition tool that performs several tasks, with the end result being that a strongly blended image is separated

into components - both in the sense that it determines the parameters for each parameter (assuming a Gaussian model) and that is physically assigns each pixel in the image to an individual object. The products of these two operations are called the component list and the component map, respectively. The fitting process (which determines the component list) and the pixel-decomposition process (which determines the component map) are designed to work cooperatively to increase the efficiency and accuracy of both. The algorithm between the decomposition is based on the function clfind described in Williams et al 1994, which uses a contouring procedure whereby a closed contour designates a separate component. The program first separates the image into clearly distint 'regions' of blended emission, then contours each region to determine the areas constituting each component and passes this information on to the fitter, which determines the component list.
The contour deblending can optionally be replaced with a simpler local maximum scan, and the fitting can be replaced with a moment-based estimation method to speed up calculations on very large images or if either primary method causes trouble, but in general this will impede the accuracy of the fit.
The function works with both two and three dimensional images.

Example

```
- im := imagemaketestimage()
- im.decompose(threshold=2.5, maxrms=1.0)
```

---

image.deconvolvecomponentlist - Function

Package	**general**
Module	**images**
Tool	**image**

Deconvolve a componentlist from the restoring beam

Synopsis
deconvolvecomponentlist(complist)

Arguments

complist	in	Componentlist to deconvolve Allowed: Componentlist tool

Returns
Componentlist tool or fail

Description

This function (short-hand name `dcl`) deconvolves a Componentlist `tool` from the restoring beam, returning a new Componentlist `tool`. If there is no restoring beam, a fail is generated.
Currently, only deconvolution of Gaussian components is supported.
See also functions setrestoringbeam and See also restoringbeam.

Example

```
- local p,m,c;
- cl := im.fitsky(p,m,c, deconvolve=F)
- cl2 := im.deconvolvecomponentlist(cl)
```

---

image.delete - Function

Package	**general**
Module	**images**
Tool	**image**

Delete the image file associated with this image tool

Synopsis
delete(done)

Arguments

done	in	Destroy this tool after deletion Allowed:T or F Default: T

Returns
T or fail

Description

This function first closes the `image tool` which detaches it from its underlying `image file`. It then deletes that `image file`. If `done=F`, the `image tool` is still viable, and can be used with function open to open a new `image file`. Otherwise the `image tool` is destroyed.

Example

```
- im := image('myimage')        # Attach to 'myimage'
- im.delete(F)                  # Close imagetool and delete 'myimage'
- im.open('myotherimage')       # Open new imagefile 'myotherimage'
```

---

image.done - Function

Package	**general**
Module	**images**
Tool	**image**

Destroy this image tool

Synopsis
done(delete)

Arguments

delete	in	Delete disk file as well ? Allowed:T or F Default: F

Returns
T or fail

Description

If you no longer need to use an `image tool`, calling this function will free up its resources. That is, it destroys the `tool`. You can no longer call any functions on the `tool` after it has been `done`.
If the Image `tool` is associated with a disk file, you can also delete that if you wish by setting `delete=T`. By default any associated disk file is not deleted.

This function is different from the `close` function because the latter does not destroy the `image tool`. For example, you can use function `open` straight after `close` on the same `tool`.

Example

```
- im := imagemaketestimage()
- im.done()
- im.open('myfile')      # Error: tool 'im' has been destroyed
```

---

image.fft - Function

Package	**general**
Module	**images**
Tool	**image**

FFT the image

Synopsis
fft(real, imag, amp, phase, axes, region, mask)

Arguments

real	in	Output real image file name Allowed:String
imag	in	Output imaginary image file name Allowed:String
amp	in	Output amplitude image file name Allowed:String
phase	in	Output phase image file name Allowed:String
axes	in	Specify pixel axes to FFT Allowed:Vector of integers Default: Sky plane(s)
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None

Returns
T or fail

Description

This function fast Fourier Transforms the image to the Fourier plane. If you leave `axes` unset, then the sky plane of the image (if there is one) is transformed. Otherwise, you can specify which axes you wish to transform. Note that if you specify a sky axis, you must specify both of them.
You specify which form you wish to see the result in by specifying the desired output image file name(s). At some point, an output complex image will also be allowed (when the Image `tool` can deal with it).
Before the FFT is taken, any masked pixels are replaced by zero. The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too.

Example

```
- myim := image('gc.small')
- myim.fft(real='r.im', amp='a.im')
```

This transforms only the sky plane(s). The real and amplitude images only are written out.

Example

```
myim := image('gc.small')
myim.fft(amp='amp.im', phase='p.im', axes=[3])
```

This transforms only the third axis of the image. The amplitude and phase images only are written out. estimate.

---

image.findsources - Function

Package	**general**
Module	**images**
Tool	**image**

Find point sources in the sky

Synopsis
findsources(nmax, cutoff, region, mask, point, width, negfind)

Arguments

nmax	in	Maximum number of sources to find Allowed:Positive Integer Default: 20
cutoff	in	Fractional cutoff level Allowed:Float Default: 0.1
region	in	The 2-D region of interest to fit Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
point	in	Find only point sources ? Allowed:T or F Default: T
width	in	Half-width of fit grid when point=F Allowed:Integer Default: 5
negfind	in	Find negative sources as well as positive ? Allowed:T or F Default: F

Returns
Componentlist tool or fail

Description

This function (short-hand name `fs`) finds strong point sources in the image. The sources are returned as a Componentlist `tool`.
An efficient method is used to locate sources under the assumption that they are point-like and not too close to the noise. Only sources with a peak greater than the `cutoff` fraction of the strongest source will be found. Only positive sources will be found, unless the `negfind=T` whereupon positive and negative sources will be found.
After the list of point sources has been made, you may choose to make a Gaussian fit for each one (`point=F`) so that shape information can be recovered as well. You can specify the half-width of the fitting grid with argument `width` which defaults to 5 (fitting grid would then be [11,11] pixels). If you set

`width=0`, this is a signal that you would still like Gaussian components returned, but a default width should be used for the Gaussian shapes. The default is such that the component is circular with a FWHM of `width` pixels. Thus, if `point=T`, the components in the returned Componentlist are Point components. If `point=F` then Gaussian components are returned.
The `region-of-interest` must be 2-dimensional and it must hold a region of the sky. Any degenerate trailing dimensions in the region are discarded.
See also the function fitsky (for which `findsources` can provide an initial estimate) and the Imagefitter `tool`.

Example

```
- myim := imagemaketestimage()
- cl := myim.findsources(nmax=5, cutoff=0.5)
- cl.edit(1)
```

All sources stronger than 0.5 of the strongest will be found. We use the Componentlist GUI to look at the strongest component.

---

image.fitallprofiles - Function

Package	**general**
Module	**images**
Tool	**image**

Fit all 1-dimensional profiles in a region

Synopsis
fitallprofiles(region, axis, mask, ngauss, poly, sigma, fit, resid)

Arguments

region	in	The region of interest Allowed:Region tool Default: Whole image
axis	in	The profile axis Allowed:Integer or unset Default: Spectral
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
ngauss	in	Number of Gaussian elements Allowed:Int Default: 1
poly	in	Order of polynomial element Allowed:Int Default: None
sigma	in	Weights image Allowed:String or unset Default: unset
fit	in	Output fitted image file name Allowed:String or unset Default: None
resid	in	Output fitted residuals image file name Allowed:String or unset Default: None

Returns
T or fail

Description
This function fits all of the 1-dimensional profiles in the specified region with a given combination of Gaussians and a polynomial. It makes a separate automatic estimate for each profile before making the fit.
Any masked input pixels do not contribute to the fit. The output mask is a copy of the default input `pixel mask` (if any); the OTF mask does not contribute. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too. Use function calcmask if you need to apply the OTF mask to the output.

Example

```
- myim := image('gc.small')
- myim.fitallprofiles (axis=3, fit='gc.fit')
```

image.fitprofile - Function

Package	**general**
Module	**images**
Tool	**image**

Fit 1-dimensional profile with functional forms

Synopsis
fitprofile(values, resid, region, axis, mask, estimate, ngauss, poly, fit, sigma)

Arguments

values	out	Output fitted or estimated profile Allowed:Numeric Glish array Default: None
resid	out	Output residual profile Allowed:Numeric Glish array
region	in	The region of interest Allowed:Region tool Default: Whole image
axis	in	The profile axis Allowed:Integer or unset Default: Spectral
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
estimate	in	Initial estimate Allowed:Record Default: Unset
ngauss	in	Number of Gaussian elements Allowed:Int Default: 1
poly	in	Order of polynomial element Allowed:Int Default: Unset
fit	in	Do fit or get estimate Allowed:T or F Default: T
sigma	in	Weights image Allowed:String or unset Default: unset

Returns
Record or fail

Description
This function fits 1-dimensional models to profiles. All data in the specified region (except along the fitting axis) are averaged.
The behaviour is primarily controlled by argument `fit`.
If `fit==T` then you need an estimate. If `estimate` is unset, then that estimate is automatically provided by fitting `ngauss` gaussians. If `estimate` is set then it is used to provide the initial estimate.
The functional form that is actually fit for will be any polynomial specified by the `poly` argument, plus the number of gaussians present in the estimate.
If `fit==F` then an estimate is created and returned. If `estimate` is empty, than an estimate for `ngauss` gaussians is made. If `estimate` is not empty, then it is just evaluated.
After calling this function, argument `values` holds the fitted or estimated profile values, and argument `resid` holds the fitted or estimated residual profile values.
The output mask is a copy of the default input `pixel mask` (if any); the OTF mask does not contribute. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too. Use function calcmask if you need to apply the OTF mask to the output.

Example

```
- myim := image('gc.small')
- local values, resid
- fit := myim.fitprofile(values, resid, axis=3, fit=T)     # Auto-est and fit
```

---

image.fitpolynomial - Function

Package	**general**
Module	**images**
Tool	**image**

Fit 1-dimensional polynomials to profiles

Synopsis
fitpolynomial(residfile, fitfile, sigmafile, axis, order, region, mask, overwrite)

Arguments

residfile	in	Output fitted residuals image file name Allowed:String or unset Default: unset
fitfile	in	Output fitted image file name Allowed:String or unset Default: unset
sigmafile	in	Input weights image for fitting Allowed:String or unset Default: unset
axis	in	The profile axis to fit Allowed:Integer or unset Default: Spectral axis if exists else last
order	in	The order of the polynomial Allowed:Integer or unset Default: 0
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
overwrite	in	Overwrite (unprompted) pre-existing output files ? Allowed:T or F Default: F

Returns
Image tool or fail

Description

This function fits 1-dimensional polynomials to profiles and subtracts the fits. You specify the order of the polynomial via the `order` argument. The returned Image tool holds the residual image. You can optionally also write out the fitted image.
The output mask is a copy of the default input `pixel mask` (if any); the OTF mask does not contribute to the output mask. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too. Use function calcmask if you need to apply the OTF mask to the output.

Example

```
- myim := image('hcn')
- myim2 := myim.fitpolynomial(order=2, axis=3)      # Image tool is virtual
```

Example

```
- myim := image('hcn')
- mask := 'indexin(3, [1:20, 30, 35, 50:60, 90:128])'        # Specify pixels to fit via mask on
- myim2 := myim.fitpolynomial(order=2, mask=mask, axis=3)    # axis 3
```

This example could be thought of, for example, as an image-based continuum substraction. You specify the axis and the pixels where the spectrum is line free (continuum), do the fit and write out the residual (continuum subtracted) image.

---

image.fitsky - Function

Package	**general**
Module	**images**
Tool	**image**

Fit 2-dimensional models to the sky

Synopsis

fitsky(pixels, pixelmask, converged, region, mask, models, estimate, fixed, includepix, excludepix, fit, deconvolve, list)

Arguments

pixels	out	Output residual pixels Allowed:Numeric Glish array
pixelmask	out	Output mask of region Allowed:Boolean Glish array
converged	out	Did the fit converge ? Allowed:T or F
region	in	The 2-D region of interest to fit Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
models	in	The model types to fit simultaneously Allowed:Vector of strings from 'gaussian' Default: gaussian
estimate	in	Initial estimate for the model Allowed:Componentlist tool Default: Internal estimate made
fixed	in	Parameters to hold fixed per model Allowed:String or vector of strings, one per model. Choose from "fxyabp" Default: All parameters vary
includepix	in	Range of pixel values to include Allowed:Vector of 1 or 2 Floats Default: See below
excludepix	in	Range of pixel values to exclude Allowed:Vector of 1 of 2 Floats Default: See below
fit	in	Do fit or get estimate Allowed:T or F Default: T
deconvolve	in	Deconvolve from restoring beam Allowed:T or F Default: F
list	in	List informative messages to the logger Allowed:T or F Default: T

Returns
Componentlist tool or fail

Description

This function fits 2-dimensional models to regions of the sky. You can fit

multiple models simulataneously, although only Gaussian models are currently implemented. The fit is returned as a Componentlist `tool`. See also the imagefitter `tool` which is an interactive high-level `tool` based around this function.

- If `fit=T` the fit is done and returned. If `fit=F` the fit is not done. Instead, the initial auto-determined estimate is returned (this is used as the starting guess of the actual fit); in this case, arguments `pixels, pixelmask, converged, estimate` and `fixed` are ignored. The auto-fit can only be determined for a single model.
- The `pixels` array is returned to you with the residual of the fit. The `pixelmask` array is returned to you with the `pixel mask` of the specified region (this reflects both the underlying `pixel mask` that you applied [if any] as well as the shape of the region).
- If the fit converged, `converged=T` on return.
- The `region-of-interest` must be 2-dimensional and it must hold a region of the sky. Any degenerate trailing dimensions in the region are discarded.
- The `models` argument specifies the simultaneous model types you would like to fit to the specified region. Currently you can only choose from 'gaussian' (minimum match). Specify the `models` argument as a vector of strings, or a string with comma or space delimiters. Thus, to fit 3 simultaneous gaussians, you would give `models='gaussian gaussian gaussian'`.
- An initial automatically determined estimate of the parameters of the model can be made for you (only for single model fits). However, if this turns out to be poor, or if you want to hold some parameters fixed, then you can enter your estimate with the `estimate` argument to which you supply a Componentlist `tool` (the same type of `tool` that is returned by this function). The estimate doesn't have to be the same type as the model you are fitting. For example you can give a Point model estimate for a Gaussian fit. The other parameter estimates will be filled in for you. The function findsources can be used to get rough point source estimates. You must supply one estimate component for every model component that you wish to fit when fitting multiple simultaneous models.
- When you fit a model, you can choose which parameters you hold fixed in the fit and which you solve for. The `fixed` argument enables you to specify which parameters of the fit you hold fixed. It is entered as a vector of strings, one string for each model specifying which parameters of the fit are fixed.

  For gaussian models, that string should contain letters chosen from

  - 'f' - hold the flux (intensity) of the model fixed
  - 'x' - hold the X location of the model fixed
  - 'y' - hold the Y location of the model fixed
  - 'a' - hold the major axis width fixed
  - 'b' - hold the minor axis width fixed
  - 'p' - hold the position angle fixed

  If you hold any parameters fixed, you may need to specify an input estimate.

- Often when fitting, it is advisable to exclude noise pixels. The `includepix` (pixel value range to include) or `excludepix` (pixel value range to exclude) arguments (you can't specify them both) allow you to select a subset of the pixels for the fit. If you only give one value for either of these vectors, say `includepix=[b]`, then this is interpreted as `includepix=[-abs(b),abs(b)]`.

  For total intensity (Stokes I) images, if neither of these are specified and the peak in the region is positive, then all pixels below zero are discarded. If the peak is negative (a negative source), all pixels above zero are discarded. For other Stokes parameters all pixels are included by default.

Example

```
- myim := image('gc.small')
- local pixels, pixelmask, converged
- cl1 := myim.fitsky(pixels, pixelmask, converged, models="gaussian")
- if (!is_fail(cl1) && converged) {
  myim2 := image('gc.big')
  cl2 := myim2.fitsky(pixels, pixelmask, converged, models="gaussian", estimate=cl1)
}
```

In this example we assume the image is 2-dimensional. We fit the first image using the internal starting estimate for the model parameters. We fit the second image using the result of the first fit as the starting estimate.

---

image.getchunk - Function

Package	**general**
Module	**images**
Tool	**image**

Get the pixel values from a regular region of the image into a Glish array

Synopsis
getchunk(blc, trc, inc, axes, list, dropdeg, getmask)

Arguments

blc	in	Bottom-Left-Corner (beginning) of pixel section Allowed:Vector of integers Default: Start of image
trc	in	Top-Right-Corner (end) of pixel section Allowed:Vector of positive integers Default: End of image
inc	in	increment (stride) along axes Allowed:Vector of positive integers Default: 1
axes	in	Axes to average over Allowed:Vector of positive integers Default: None
list	in	List bounding box to logger ? Allowed:T or F Default: F
dropdeg	in	Drop degenerate axes ? Allowed:T or F Default: F
getmask	in	Get the pixel mask as well Allowed:T or F Default: F

Returns
Glish array, Record, or fail

Description

This function returns the pixels (and opionally the pixel mask) from the `image file` between `blc` and `trc` inclusively. An increment may be specified with `inc`. Note that if you retrieve too many pixels, you might cause swapping since the pixels are kept in memory.
Any illegal `blc` values are set to unity. Any illegal `trc` values are set to the end of the image. If any `trc` $<$ `blc`, you get the whole image for that axis. Any illegal `inc` values are set to unity.
The argument `axes` can be used to reduce the dimensionality of the output array. It specifies which pixel axes of the image **average** the data over. For

example, consider a 3-D image. With `axes=[1,2]` and all other arguments left at their defaults, the result would be a 1-D vector, a profile along the third axis, with the data averaged over the first two axes.
A related function is getregion which retrieves the pixels and `pixel mask` from a potentially more complex `region-of-interest`. Function `getchunk` is retained because it is faster and therefore preferable for repeated operation in loops if the `pixel mask` is not required and the region is a simple box.
If you ask for the pixel mask as well, the return value is a record holding two fields, 'pixel' and 'pixelmask'. If you do not ask for the pixel mask (default), the pixels are returned directly.

Example
Suppose that we have a 3-dimensional image called *im*. Then:

```
- pix := im.getchunk()                        # all pixels
- pix := im.getchunk([1,1,1], [10,10,1])      # 10 by 10 section of plane # 1
- pix := im.getchunk([1,1], [1,1])            # first spectrum
- pix := im.getchunk(inc=[1,5])               # all planes, decimated by 5 in y
#
- rec := im.getchunk(getmask=T)               # Get pixels and pixelmask
- print field_names(rec)                      # into named fields of record
pixel pixelmask
```

---

image.getregion - Function

Package	**general**
Module	**images**
Tool	**image**

Get pixels and mask from a region-of-interest of the image

Synopsis
getregion(pixels, pixelmask, region, axes, mask, list, dropdeg)

Arguments

pixels	out	The recovered pixel values Allowed:Glish array
pixelmask	out	The recovered pixel mask values Allowed:Glish array
region	in	The region of interest Allowed:Region tool Default: Whole image
axes	in	Axes to average over Allowed:Vector of positive integers Default: None
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
list	in	List the bounding box to the logger Allowed:T or F Default: F
dropdeg	in	Drop degenerate axes Allowed:T or F Default: F

Returns
T or fail

Description

This function recovers the image pixel and `pixel mask` values in the given region-of-interest. You can recover either or both of these by specifying the appropriate argument. Regardless of the shape of the `region` you have specified, the shape of the `pixels` and `pixelmask` arrays must necessarily be the bounding box of the specified region. If the region extends beyond the image, it is truncated.
Recall that the recovered `pixel mask` will reflect both the `pixel mask` stored in the image, and the `region-of-interest` (their masks are 'anded') – see the discussion in the introduction about this.
The argument `axes` can be used to reduce the dimensionality of the output array. It specifies which pixel axes of the image average the data over. For example, consider a 3-D image. With `axes=[1,2]` and all other arguments left at their defaults, the result would be a 1-D vector, a profile along the third axis, with the data averaged over the first two axes.
This function differs in three ways from `getchunk`. First, the region can be much more complex (e.g. a union of polygons) than the simple `blc`, `trc`, and `inc` of `getchunk` (although such a region can be created of course). Second, it recovers the `pixel mask` as well as the pixels. Third, it is less efficient than `getchunk` for doing the same thing as `getchunk`. So if you are interested in

say, iterating through an image, getting a regular hyper-cube of pixels and doing something with them, then `getchunk` will be faster. This would be especially noticeable if you iterated line by line through a large image.

Example
Suppose that we have a 3-dimensional image called *cube* and wish to recover the pixel and `pixel mask` values from a simple regular region.

```
- im := image('cube')
- r1 := drm.box(blc=[10,10,10], trc=[30,40])  # Create region
- local pixels, pixelmask
- im.getregion(pixels, pixelmask, r1)
```

Example

```
- im := image('cube')
- local mm, pp
- im.getregion(pixels=pp)
- im.getregion(pixelmask=mm)
```

In these examples we recover first the pixels only and then the pixel mask only.

---

image.getslice - Function

Package	**general**
Module	**images**
Tool	**image**

Get 1-D slice from the image

Synopsis
getslice(x, y, axes, coord, npts, method, plot)

Arguments

x	in	Polyline x vertices in absolute pixel coordinates Allowed: Vector of float
y	in	Polyline y vertices in absolute pixel coordinates Allowed: Vector of float
axes	in	Pixel axes of plane holding slice Allowed: Vector of integers Default: First two axes
coord	in	Specify pixel coordinate for other axes Allowed: Vector of integers Default: Unity
npts	in	Number of points in slice Allowed: Integer Default: Auto determination
method	in	The interpolation method Allowed: String from 'nearest', 'linear', 'cubic' Default: 'linear'
plot	in	Make a plot of the slice Allowed: T or F Default: F

Returns
Glish Record, or fail

Description

This function returns a 1-D slice (the pixels and opionally the pixel mask) from the `image file`. The slice is constrained to lie in a plane of two cardinal axes (e.g. XY or YZ). At some point this constraint will be relaxed. A range of interpolation schemes are available.
You specify the slice as a polyline giving the x (`x`) and y (`y` coordinates and the axes of the plane holding that slice (`axes`). As well, you must specify the absolute pixel coordinates of the other axes (`coord`). This defaults to the first pixel (e.g. first plane).
The return value is a Glish record with fields 'pixels' (interpolated intensity), 'mask' (interpolated mask), 'xpos' (x-location in absolute pixel coordinates), 'ypos' (y-location in absolute pixel coordinates), 'distance' (distance along slice in pixels), 'axes' (the x and y axes of slice).
You can optionally plot the slice via argument `plot` which defaults to False.

Example
Suppose that we have a 3-dimensional image called *im*. Then:

```
- im := imagemaketestimage();
```

```
- rec := im.getslice (x=[1,20], y=[2,30])            # SLice from [1,2] -> [20,30]
- print field_names(rec)
pixel mask xpos ypos distance axes
-
- rec := im.getslice (x=[1,20,25,11], y=[2,30,32,40]) # Polyline slice
```

---

image.hanning - Function

Package	**general**
Module	**images**
Tool	**image**

Convolve one axis of image with a Hanning kernel

Synopsis
hanning(outfile, region, mask, axis, drop, overwrite, async)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
region	in	Region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
axis	in	Axis to convolve Allowed:Positive integer Default: Spectral axis if there is one
drop	in	Drop every other pixel on output ? Allowed:Bool Default: T
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
Image tool or fail

Description

This function performs Hanning convolution
($z[i] = 0.25 * y[i-1] + 0.5 * y[i] + 0.25 * y[i+1]$) of one axis of an image. The convolution is done in the image domain (i.e., not with an FFT). You can optionally drop every other pixel on output. If you do this, the first output pixel is centred on the second input pixel.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
Masked pixels will be assigned the value 0.0 before convolution. The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too.
See also the other convolution functions convolve2d, sepconvolve and convolve.

Example

```
- dowait := T
- im := image('x')
- im2 := im.hanning(outfile='x.hann')
```

This would Hanning smooth the spectral axis of the image associated with the `image tool` **im**, if there is one. Every other pixel on the convolution axis is dropped on output, by default.

---

image.haslock - Function

Package	**general**
Module	**images**
Tool	**image**

Does this image have any locks set ?

Synopsis
haslock()

Returns
Vector of Bool or fail

Description

This function can be used to find out whether the image has a read or a write lock set. It is not of general user interest. It returns a vector of Booleans of length 2. Position 1 says whether a read lock is set, position 2 says whether a write lock is set.
In general locking is handled automatically, with a built in lock release cycle. However, this function can be useful in scripts when a file is being shared between more than one process. See also functions unlock and lock.

Example

```
- im := image('xx')
- im.lock(write=T)
- im.haslock()
[T T]
- im.unlock()
- im.haslock()
[F F]
- im.lock(F)
- im.haslock()
[T F]
```

This acquires a read/write lock on the file and then unlocks it and acquires just a read lock.

---

image.histograms - Function

Package	**general**
Module	**images**
Tool	**image**

Compute histograms from the image

Synopsis
histograms(histout, axes, region, mask, nbins, includepix, gauss, cumu, log, list, plotter, nx, ny, size, force, disk, async)

Arguments

histout	out	Glish record containing the histograms Allowed: Glish variable Default: No retrieval
axes	in	List of axes to compute histograms over Allowed: Vector of integers Default: All axes
region	in	Region of interest Allowed: Region tool Default: Whole image
mask	in	OTF mask Allowed: Boolean LEL expression or mask region Default: None
nbins	in	Number of bins in histograms Allowed: Positive integer Default: 25
includepix	in	Range of pixel values to include Allowed: A vector of 1 or 2 Floats Default: Include all pixels
gauss	in	If T overlay a Gaussian on each histogram Allowed: Bool Default: F
cumu	in	If T plot cumulative histograms, otherwise plot non-cumulatively Allowed: Bool Default: F
log	in	If T plot the ordinate logarithmically, otherwise plot linearly Allowed: Bool Default: F
list	in	If T then list some statistics as well Allowed: Bool Default: T
plotter	in	The PGPLOT device name to make plots on Allowed: Any valid PGPLOT device Default: No plotting
nx	in	The number of subplots per page in the x direction Allowed: Positive integer Default: 1
ny	in	The number of subplots per page in the y direction Allowed: Positive integer Default: 1
size	in	Size of plotter Allowed: Vector of integers Default: [600,450]
force	in	If T then force the stored statistical accumulations to be regenerated Allowed: Bool Default: F
disk	in	If T then force the storage image to disk Allowed: Bool Default: F
async	in	Run asynchronously? Allowed: Bool Default: !dowait - but always F if plotting

Returns
T or fail

Description

This function (short-hand name is `histo`) computes histograms of the pixel values in the `image file`. You can plot them and retrieve them (into a *Glish* record) for further analysis or display.
The chunk of the image over which you compute the histograms is specified by a vector of axis numbers (argument `axes`). For example, consider a 3-dimensional image for which you specify `axes=[1,3]`. The histograms would be computed for each XZ (axes 1 and 3) plane in the image. You could then examine those histograms as a function of the Y (axis 2) axis. Or perhaps you set `axes=[3]`, whereupon you could examine the histogram for each Z (axis 3) profile as a function of X and Y location in the image.
You have control over the number of bins for each histogram (`nbins`). The bin width is worked out automatically for each histogram and may vary from histogram to histogram (the range of pixel values is worked out for each chunk being histogrammed).
You have control over which pixels are included in the histograms via the `includepix` argument. This vector specifies a range of pixel values to be included in the histograms. If you only give one value for this, say `includepix=[b]`, then this is interpreted as `includepix=[-abs(b),abs(b)]`. If you specify an inclusion range, then the range of pixel intensities over which the histograms are binned is given by this range too. This is a way to make the bin width the same for each histogram.
You have control over the form of the histogram; the ordinate can be linear or logarithmic (`log=T`), and cumulative or non-cumulative (`cumu=T`).
You can overlay a Gaussian on each histogram (`gauss=T`). It has the same mean and standard deviation about the mean of the pixels that were binned and the same integral as the histogram.
You can list some additional statistical information about the data in each image chunk that is being histogrammed (`list=T`).
You can retrieve the histograms into a *Glish* record (the counts and the abcissa values for each histogram). The names of the fields in the record are `'values'` and `'counts'`. The shape of the first dimension of those arrays contained in those fields is `nbins`. The number and shape of the remaining dimensions are those in the image for which you did not compute the histograms. For example, in the second example above, we set `axes=[3]` and asked for histograms as a function of the remaining axes, in this case, the X and Y (axes 1 and 2) axes. The shape of each histogram array is then [nbins,nx,ny]. The form of the retrieved histograms is that specified by the `cumu` and `log` arguments.
The plotting is done directly on a PGPLOT plotting device (function is always run synchronously when plotting). The syntax is `plotter=name/type`. For

example `plotter='plot1.ps/ps'` (disk postscript file) or
`plotter='plot1/glish'` (Glish PGplotter).
This function generates "storage" images, into which the histograms and statistics are written. They are only regenerated when necessary. For example, if you run the function twice successively with identical arguments, the histograms will be directly retrieved from the storage image. However, you can force regeneration of the storage image if you set `force=T`. The storage medium is either in memory or on disk, depending upon its size. You can force it to disk if you set `disk=T`, otherwise it decides for itself.

Example

```
- im.histograms(plotter='plot/glish')
```

In this example we plot the histogram of the entire image on a PGPLOT X-windows device.

Example

```
- im.histograms(histout=myhist,axes=[1,3],nbins=30,includepix=1e-3,
                cumu=T,gauss=T,plotter='plot/glish',nx=4,ny=3)
```

In this example, let us assume the image has 3 dimensions. We generate cumulative histograms of XZ (axes 1 and 3) planes for pixels with values in the range -0.001 to 0.001 and plot them with Gaussian overlays as a function of Y-axis location on the standard PGPLOT X-windows device with 12 plots per page. The histograms are retrieved into a *Glish* record called `myhist` for further use (because we are plotting the function is run synchronously).

---

image.history - Function

Package	**general**
Module	**images**
Tool	**image**

Recover and/or list the history file

Synopsis
history(list, browse)

Arguments

list	in	List history to the logger ? Allowed: T or F Default: F
browse	in	Browse history with table browser? Allowed: T or F Default: T

Returns
Vector of strings, T, or fail

Description

This function allows you to access the history file.
By default, you browse it with the Table browser (`browse=T`) and the return value is T or a fail.
If `browse=F`, then if `list=F`, the history is returned by the function as a vector of strings. If `list=T`, the history is sent to the logger and the function returns T or a fail.
Currently, AIPS++ does not write new history information (although you can set directly set a new history file with the function sethistory). Only that which has come from FITS file conversions will be stored and listable here.

Example

```
- im.history();                       # Browse
- h := im.history(list=F, browse=F)   # Recover history in variable h
- im.history(list=T, browse=F)        # List  history to logger
```

---

image.id - Function

Package **general**
Module **images**
Tool **image**

Return the fundamental identifier of this tool

Synopsis
id()

Description

This function should be of little interest to users. It returns the fundamental identifier of the tool.

Returns
ToolID - a record

Example

```
- id := im.id()
```

---

image.insert - Function

Package	**general**
Module	**images**
Tool	**image**

Insert specified image into this image

Synopsis
insert(infile, region, locate)

Arguments

infile	in	Image file name or Image tool to insert Allowed:String Default: unset
region	in	The region of interest of the input image Allowed:Region tool Default: Whole image
locate	in	Location of input image in output image Allowed:Vector Default: Centrally located

Returns
Image tool or fail

Description

This function inserts the specified image (or part of it) into this image. The specified image may be given via argument `infile` as a disk file name (it may be in native AIPS++, FITS, or Miriad format; Look here for more information on foreign images) or directly as an Image tool.
If the `locate` vector is not given, then the images are aligned (to an integer pixel shift) by their reference pixels.
If `locate` vector is given, then those values that are given, give the absolute pixel in the output (this) image of the bottom left corner of the input (sub)image. For those values that are not given, the input image is symmetrically placed in the output image.

Example

```
- dowait := T
- im := imagefromshape(shape=[200,200])
- im.insert(infile='myfile.insert')              # Align by reference pixel
- im.set(0.0)
- im2 := image('myfile.insert')
- im.insert(infile=im2, locate=[])               # Align centrally, specify as Image tool
- im.set(0.0)
- im.insert(infile='myfile.insert', locate=[20]) # Align axis 1 as given and axis 2 centrally
```

---

image.isopen - Function

Package	**general**
Module	**images**
Tool	**image**

Is this Image `tool` open ?

Synopsis
isopen()

Returns
Bool

Description

This function can be used to find out whether the Image `tool` is open or not.

Example

```
- im := imagemaketestimage('zz')
- im.isopen()
T
- im.close()
- im.isopen()
F
- im.open('zz')
F
- im.isopen()
T
```

---

image.ispersistent - Function

Package	**general**
Module	**images**
Tool	**image**

Is the image persistent ?

Synopsis
ispersistent()

Returns
Bool or fail

Description

This function can be used to find out whether the image is persistent on disk or not. There is a subtle difference from the image being virtual. For example, a virtual image which references another which is on disk is termed persistent.

Example

```
- im := imagefromshape(outfile='tmp', shape=[10,20])
- im.ispersistent()
T
- im2 := imagefromimage(infile='tmp', region=drm.box())
- im2.ispersistent()
F
- im3 := im2.subimage()
- im3.ispersistent()                  # Persistent virtual image !
T
- im4 := imagecalc(pixels='tmp+tmp2')
- im4.ispersistent()
F
```

---

image.lock - Function

Package	**general**
Module	**images**
Tool	**image**

Acquire a lock on the image

Synopsis
lock(write, nattempts)

Arguments

write	in	Acquire a read/write (T) or a readonly (F) lock Allowed:T or F Default: T
nattempts	in	Number of attempts Allowed:Non-negative integer Default: Forever

Returns
T or fail

Description

This function can be used to acquire a Read or a Read/Write lock on the `image file`. It is not of general user interest.

In general locking is handled automatically, with a built in lock release cycle. However, this function can be useful in scripts when a file is being shared between more than one process. See also functions unlock and haslock.

Example

```
- im := image('xx')
- im.lock(write=T)
- im.unlock()
- im.lock(write=F)
```

This acquires a read/write lock on the file. Then we unlock it and acquire a readonly lock.

---

image.makecomplex - Function

Package	**general**
Module	**images**
Tool	**image**

Make a complex image

Synopsis
makecomplex(outfile, imag, region, overwrite)

Arguments

outfile	in	Output Complex (disk) image file name Allowed:String
imag	in	Imaginary image file name Allowed:String
region	in	The region of interest Allowed:Region tool Default: Whole image
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
T or fail

Description

This function combines the current image with another image to make a complex image. The current image (i.e. that associated with this Image `tool` is assumed to be the Real image). You supply the Imaginary image; it must be disk-based at this time.
The output image cannot be associated with an Image `tool` (does not handle Complex images yet) and so the best you can do is write it to disk. The Viewer can view it.

Example

```
- myim := image('im.real')
- myim.makecomplex('im.complex', 'im.imag')
```

---

image.maskhandler - Function

Package **general**
Module **images**
Tool **image**

Handle pixel masks

Synopsis
maskhandler(op, name)

Description

Arguments

op	in	The operation Allowed: String, one of 'set', 'default', 'delete', 'rename', 'get', 'copy'
name	in	Name of mask or masks Allowed: Vector of strings

Returns
Vector string, string, T or fail

This function (short-hand name `mh`) is used to manage or handle `pixel masks` (there is also a custom GUI interface through the maskhandlergui function). An AIPS++ image may contain zero, one or more `pixel masks`. Any of these masks can be designated the default `pixel mask`. The default mask is acted upon by AIPS++ applications. For example, if you ask for statistics from an image, pixels which are masked as bad (F) will be excluded from the calculations.
This function has an argument (`op`) that specifies the behaviour. In all cases, you can shorten the operation string to three characters. It is not the job of this function to modify the values of masks.

- default - this retrieves the name of the default `pixel mask` as the return value of the function call.
- get - this retrieves the name(s) of the existing `pixel masks` as the return value of the function call (string or vector of strings).
- set - this lets you change the default `pixel mask` to that given by the `name` argument. If `name` is empty, then the default mask is unset (i.e. an all good mask is effectively applied).
- delete - this lets you delete the `pixel masks` specified by the `name` argument. To delete more than one mask, `name` can be a vector of strings. Any supplied `pixel mask` name that does not exist is silently ignored.
- rename - this lets you rename the mask specified by `name[1]` to `name[2]`. Thus the `name` argument must be a vector of length 2.
- copy - this lets you copy a mask to another in the same image, or copy a mask from another image into this image. Thus the `name` argument must be a vector of length 2.

  For the first case, the first element of `name` must be the name of the mask to copy, and the second element must be the name of the `pixel mask` to which it will be copied.

  For the second case, the first element of `name` must be the name of the input image and `pixel mask` with a colon delimiter (e.g. `hcn:mask2`). The second element must be the name of the `pixel mask` to which the input `pixel mask` will be copied.

Use the summary function to see the available `pixel masks`. You can do this either via the logger display, or via the returned record, which contains the mask names. In the logger display, any `pixel mask` which is not the default mask is listed in square brackets. If a default mask is set, it is listed first, and is not enclosed in square brackets.

Example

```
- im := image('myimage')
- local names, name
- names := im.maskhandler('get')            # Get the mask names
- name := im.maskhandler('default')         # Get the default mask names
- im.maskhandler('set', 'mask1')            # Make 'mask1' the default mask
- im.maskhandler('set', '')                 # Unset the default mask
- im.maskhandler('delete', 'mask1')         # Delete 'mask1'
- im.maskhandler('delete', 'mask0 mask1')   # Delete 'mask0' and 'mask1'
- im.maskhandler('rename', 'mask1 mask0')   # Rename 'mask1' to 'mask0'
- im.maskhandler('copy', 'g1.app:mask1 mask10')   # Copy 'mask1' from image 'g1.app'  to 'mask10'
```

---

image.maskhandlergui - Function

Package	**general**
Module	**images**
Tool	**image**

Handle masks using a GUI interface

Synopsis
maskhandlergui(parent, widgetset)

Arguments

parent	in	Parent frame to embed GUI in Allowed: *Glish*Tk agent Default: Create stand alone
widgetset	in	The widgetset to use for the GUI Allowed: A widgetserver tool Default: dws

Returns
Agent or fail

Description

General This function (short-hand name `mhgui`) starts the custom GUI interface for the maskhandler function. Here is an example of the GUI.
The left-hand list box shows you the `pixel masks` currently available in the image. Underneath the list box the default `pixel mask`, if any, is listed. You

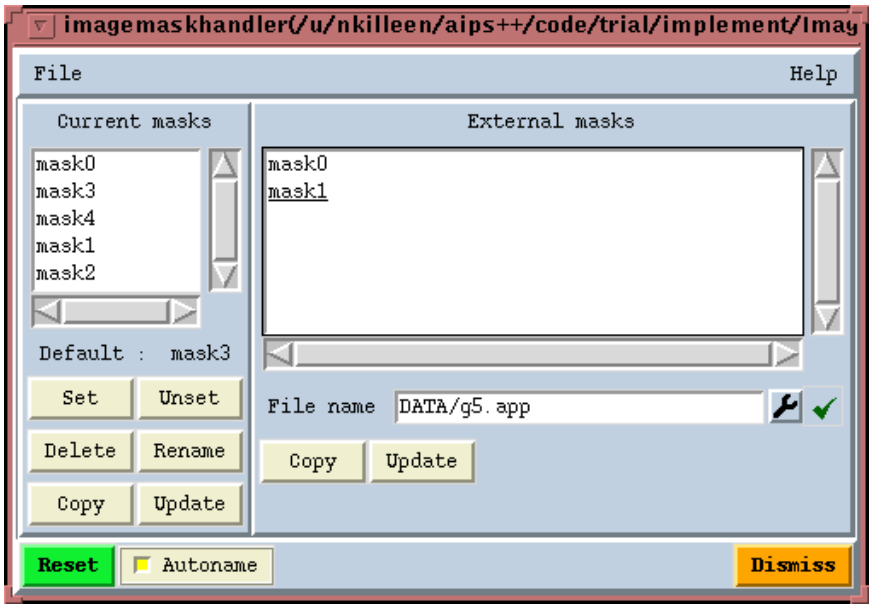

Figure 1.1: The maskhandler GUI

have buttons under the left-hand list box to apply the desired operation. The sequence is to select a `pixel mask` or `pixel masks` from the list box, and then click the desired button. Popuphelp is on each button to tell you what they do. The right-hand list box is used to list the `pixel masks` from some other image. You enter the `image file` name with the file entry widget and then enter carriage-return. You can copy masks from that `image file` to the current image.
If you don't provide a parent frame (the `parent` argument), the GUI is made in a standalone mode. This means it creates go, reset and dismiss buttons. If you provide the parent frame, you must also provide these buttons and act on them.

Recovering the GUI
When you run this function, it returns an agent which is actually a reference to the underlying GUI. In addition, the GUI is left on the screen. You can explicitly dismiss it by pressing the Dismiss button or selecting Dismiss from the File menu. To recover the GUI you can do one of two things.

1. Run the command `maskhandlergui` again. This will return another reference of the GUI to you. However, it will reset some parts of the GUI because it is possible that the image changed since you last ran this function (the `pixel masks` might have changed, it might even be a different `image file` attached to this `tool`).

Thus

```
- im := image('myimage')
- g := im.maskhandlergui()            # Click Dismiss on GUI
- g := im.maskhandlergui()            # Get GUI back with some reset
```

2. Apply the `gui` function to the reference returned when you ran the `maskhandlergui` function. This simply recovers the GUI with no resetting. If the image **has** changed, this will not reflect it.

```
- im := image('myimage')
- g := im.maskhandlergui()            # Click Dismiss on GUI
- g.gui()                             # Get GUI back with no change
```

If you destroy the GUI (by selecting Done from the File menu) then you must run the momentsgui function again to regenerate it.

---

image.miscinfo - Function

Package	**general**
Module	**images**
Tool	**image**

Get the miscellaneous information record from an image

Synopsis
miscinfo()

Returns
Glish record or fail

Description

An AIPS++ `image file` can accumulate various miscellaneous information during its lifetime; it is stored in a *Glish* record called the `miscinfo` record. For example, the FITS filler puts header keywords it doesn't otherwise use into the `miscinfo` record. This `miscinfo` record is not guaranteed to have any entries, so it's up to you to check for any fields that you require.
You can also put things into this record (see setmiscinfo) yourself, to keep information that the system might not otherwise store for you.

When the image is written out to FITS, the items in the `miscinfo` record are written to the FITS file as keywords with the corresponding record field name.

Example

```
- im.miscinfo()             # print the record
[bscale=1, bzero=0, cellscal=CONSTANT]
- header := im.miscinfo()   # capture the record for further use
```

---

image.modify - Function

Package	**general**
Module	**images**
Tool	**image**

Modify image with a model

Synopsis
modify(model, region, mask, subtract, list, async)

Arguments

model	in	The model Allowed:Componentlist tool
region	in	The 2-D region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
subtract	in	Subtract or add the model Allowed:T or F Default: T
list	in	List informative messages to the logger Allowed:T or F Default: T
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
T or fail

Description

This function applies a model of the sky to the image. You can add or subtract the model which is contained in a Componentlist `tool`.
The pixel values are only changed where the total mask (combination of the default `pixel mask` [if any] and the OTF mask) is good (True). If the computation fails for a particular pixel (e.g. coordinate undefined) that pixel will be masked bad.

Example

```
- myim := image('gc.small')
- cl := componentlist('list')
- myim.modify(cl)
```

In this example we subtract the model contained in the Table called *list*.

---

image.maxfit - Function

Package	**general**
Module	**images**
Tool	**image**

Find maximum and do parabolic fit in the sky

Synopsis
maxfit(region, point, width, negfind, list)

Arguments

region	in	The 2-D region of interest to fit Allowed:Region tool Default: Whole image
point	in	Find only point sources ? Allowed:T or F Default: T
width	in	Half-width of fit grid when point=F Allowed:Integer Default: 5
negfind	in	Find negative sources as well as positive ? Allowed:T or F Default: F
list	in	List the fitted parameters to the logger ? Allowed:T or F Default: T

Returns
Componentlist tool or fail

Description

This function finds the pixel with the maximum value in the region, and then uses function findsources to generate a Componentlist with one component. The component will be of type Point (`point=T`) or Gaussian (`point=F`).
If `negfind=F` the maximum pixel value is found in the region and fit. If `negfind=T` the absolute maximum pixel value is found in the region and fit.
See function findsources for a description of arguments `point` and `width`.
See also the function fitsky.

Example

```
- myim := imagemaketestimage()
- cl := myim.maxfit()
- cl.edit(1)                  # There is only one component
```

---

image.moments - Function

Package **general**
Module **images**
Tool **image**

Compute moments from an image

Synopsis
moments(moments, axis, region, mask, method, smoothaxes, smoothtypes, smoothwidths, includepix, excludepix, peaksnr, stddev, doppler, outfile, smoothout, plotter, nx, ny, yind, overwrite, drop, async)

Arguments

moments	in	List of moments that you would like to compute Allowed: Vector of integers Default: 0 (integrated spectrum)
axis	in	The moment axis Allowed: Positive integer Default: The spectral axis if there is one
region	in	Region of interest Allowed: Region tool Default: Whole image
mask	in	OTF mask Allowed: Boolean LEL expression or mask region Default: None
method	in	List of windowing and/or fitting functions you would like to invoke Allowed: Vector of strings from 'window', 'fit', and 'interactive' Default: Don't invoke the window or fit functions, and don't invoke any interactive functions
smoothaxes	in	List of axes to smooth Allowed: Vector of integers Default: No smoothing
smoothtypes	in	List of smoothing kernel types, one for each axis to smooth Allowed: Vector of strings from 'gauss', 'boxcar', 'hanning' Default: No smoothing
smoothwidths	in	List of widths (full width for boxcar, full width at half maximum for gaussian, 3 for Hanning) in pixels for the smoothing kernels Allowed: Vector of numeric, quantity or string Default: No smoothing
includepix	in	Range of pixel values to include Allowed: Vector of 1 or 2 Floats Default: Include all pixels

excludepix	in	Range of pixel values to exclude Allowed: Vector of 1 of 2 Floats Default: Exclude no pixels
peaksnr	in	The SNR ratio below which the spectrum will be rejected as noise (used by the window and fit functions only) Allowed: A positive float Default: 3
stddev	in	Standard deviation of the noise signal in the image (used by the window and fit functions only) Allowed: Positive float Default: Automatic noise determination
doppler	in	Velocity doppler definition for velocity computations along spectral axes Allowed: String Default: RADIO
outfile	in	Output image file name (or root for multiple moments) Allowed: String Default: input + an auto-determined suffix
smoothout	in	Output file name for convolved image Allowed: String Default: Don't save the convolved image
plotter	in	The PGPLOT device name to make plots on Allowed: Any valid PGPLOT device Default: No plotting
nx	in	The number of subplots per page in the x direction Allowed: Any positive integer Default: 1
ny	in	The number of subplots per page in the y direction Allowed: Any positive integer Default: 1
yind	in	Scale the y axis of the profile plots independently Allowed: Bool Default: False
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed: T or F Default: F
drop	in	Drop moments axis from output images ? Allowed: T or F Default: T
async	in	Run asynchronously? Allowed: Bool Default: !dowait - but always F if plotting

Returns
Image tool or fail

Description

Summary
The primary goal of this function is to enable you to analyze a multi-dimensional image by generating moments of a specified axis. This is a time-honoured spectral-line analysis technique used for extracting information about spectral lines. There is also a custom GUI interface available via the momentsgui function as the command line interface is presently rather cumbersome.
You can generate one or more output moment images. The return value of this function is an Image `tool` holding the **first** of the output moment images.
The word 'moment' is used loosely here. It refers to collapsing an axis (the moment axis) to one pixel and setting the value of that pixel (for all of the other non-collapsed axes) to something computed from the data values along the moment axis. For example, take an RA-DEC-Velocity cube, collapse the velocity axis by computing the mean intensity at each RA-DEC pixel. This function offers many different moments and a variety of interactive and automatic methods to compute them.
We try to make a distinction between a 'moment' and a 'method'. This boundary is a little blurred, but it claims to refer to the distinction between what you are computing, and how the pixels that were included in that computation were selected. For example, a 'moment' would be the average value of some pixel values in a spectrum. A 'method' for selecting those pixels would be a simple pixel value range specifying which pixels should be included.
There are many available moments, and you specify each one with an integer code as it would get rather cumbersome to refer to them via strings. In the list below, the value of the ith pixel of the spectrum is I_i, the coordinate of this pixel is v_i (of course it may not be velocity), and there are n pixels in the spectrum. The available moments are:

- -1 – the mean value of the spectrum

$$\frac{1}{n}\sum I_i$$

- 0 – the integrated value of the spectrum

$$M_0 = \Delta v \sum I_i$$

  where Δv is the width (in world coordinate units) of a pixel along the moment axis

- 1 – the intensity weighted coordinate (this is traditionally used to get 'velocity fields')

$$M_1 = \frac{\sum I_i v_i}{M_0}$$

- 2 – the intensity weighted dispersion of the coordinate (this is traditionally used to get 'velocity dispersion fields')

$$\sqrt{\frac{\sum I_i \left(v_i - M_1\right)^2}{M_0}}$$

- 3 – the median of I

- 4 – the median coordinate. Here we treat the spectrum as a probability distribution, generate the cumulative distribution, and then find the coordinate corresponding to the 50% value. This moment is not very robust, but it is useful for quickly generating a velocity field in a way that is not sensitive to noise. However, it will only give sensible results under certain conditions. The generation of the cumulative distribution and the finding of the 50% level really only makes sense if the cumulative distribution is monotonic. This essentially means only selecting pixels which are positive or negative. For this reason, this moment type is only supported with the basic method (see below – i.e. no smoothing, no windowing, no fitting) with a pixel selection range that is either all positive, or all negative

- 5 – the standard deviation about the mean of the spectrum

$$\sqrt{\frac{1}{(n-1)} \sum \left(I_i - \bar{I}\right)^2}$$

- 6 – the root mean square of the spectrum

$$\sqrt{\frac{1}{n} \sum I_i^2}$$

- 7 – the absolute mean deviation of the spectrum

$$\frac{1}{n}\sum|(I_i - \bar{I})|$$

- 8 – the maximum value of the spectrum
- 9 – the coordinate of the maximum value of the spectrum
- 10 – the minimum value of the spectrum
- 11 – the coordinate of the minimum value of the spectrum

Smoothing
The purpose of the smoothing functionality is purely to provide a mask. Thus, you can smooth the input image, apply a pixel include or exclude range, and generate a smoothed mask which is then applied before the moments are generated. The smoothed data are not used to compute the actual moments; that is always done from the original data.

Basic Method
The basic method is to just compute moments directly from the pixel values. This can be modified by applying pixel value inclusion or exclusion ranges (arguments `includepix` and `excludepix`).
You can then also convolve the image (arguments `smoothaxes`, `smoothtypes`, and `smoothwidths`) and find a mask based on the inclusion or exclusion ranges applied to the convolved image. This mask is then applied to the unsmoothed data for moment computation.

Window Method
The window method (invoked with argument `method='window'`) does no pixel-value-based selection. Instead a window is found (hopefully surrounding the spectral line feature) and only the pixels in that window are used for computation. This window can be found from the convolved or unconvolved image (arguments `smoothaxes`, `smoothtypes`, and `smoothwidths`).
The moments are always computed from the unconvolved data. The window can be found (for each spectrum) interactively or automatically. The automatic methods are via Bosma's converging mean algorithm (`method='window'`) or by fitting Gaussians and taking $\pm 3\sigma$ as the window (`method='window,fit'`). The interactive methods are a direct specification of the window with the cursor (`method='window,interactive'`) and by interactive fitting of Gaussians and taking ±3-sigma as the window (`method='window,fit,interactive'`).

In Bosma's algorithm, an initial guess for a range of pixels surrounding a spectral feature is refined by widening until the mean of the pixels outside of the range converges (to the noise).

Fit Method
The fit method (`method='fit'`) fits Gaussians to spectral features either automatically or interactively (`method='fit,interactive'`). The moments are then computed from the Gaussian fits (not the data themselves).
The interactive methods are very user intensive. You have to do something for each spectrum. These are really only useful for images with a manageably small number of spectra.

Other Arguments

- `outfile` - If you are creating just one moment image, and you specify `outfile`, then the image is created on disk with this name. If you leave `outfile` empty then a temporary image is created. In both cases, you can access this image with the returned Image `tool`. If you are making more than one moment image, then theses images are always created on disk. If you specify `outfile` then this is the root for the output file names. If you don't specify it, then the input image name is used as the root.

- `smoothing` - If you smooth the image to generate a mask, you specify the kernel widths via the `smoothwidths` argument in the same way as in the sepconvolve function. See it for details.

- `plotter` - The plotting is done directly on a PGPLOT plotting device. The syntax is `plotter=name/type`. For example `plotter='plot1.ps/ps'` (disk postscript file) or `plotter='plot/glish'` (*Glish* PGplotter).

  Note also that if you specify a plotting device but do not ask for an interactive option, then plots will be made showing you the effect of the algorithm on each spectrum. For example, you may have selected the automatic windowing method. If you have specified a plotting device, then each spectrum will be plotted and the window marked on each plot.

- `stddev` - Some of the automatic methods also require an estimate of the noise level in the image. This is used to assess whether a spectrum is purely noise or not, and whether there is any signal worth digging out. If you don't give it via the `stddev` argument, it will be worked out automatically from a Gaussian fit to the bins above 25% from a histogram of the entire image. If you have specified the plotting device as well, you get the chance to interact with this fitting process.

- `includepix, excludepix` - The vectors given by arguments `includepix` and `excludepix` specify a range of pixel values for which pixels are

either included or excluded. They are mutually exclusive; you can specify one or the other, but not both. If you only give one value for either of these, say `includepix=b`, then this is interpreted as `includepix=[-abs(b),abs(b)]`.

The convolving point-spread function is normalized to have a volume of unity. This means that point sources are depressed in value, but extended sources that are large with respect to the PSF remain essentially on the same intensity scale; these are the structures you are trying to find with the convolution so this is what you want. If you convolve the image, then arguments like `includepix` select based upon the convolved image pixel values. If you are having trouble getting these right, you can output the convolved image (`smoothout`) and assess the validity of your pixel ranges. Note also that if you are Hanning convolving (usually used on a velocity axis), then the width for this kernel must be 3 pixels (triangular smoothing kernels of other widths have no valid theoretical basis).

- `doppler` - If you compute the moments along a spectral axis, it is conventional to compute the world coordinate (needed for moments 0, 1 and 2) along that axis in "km/s". The argument `doppler` lets you specify what doppler convention the velocity will be calculated in. You can choose from `doppler=radio, optical, true`. See function summary for the definitions of these codes. For other moment-axis types, the world coordinate is computed in the native units.

- `mask` - The total input mask is the combination of the default `pixel mask` (if any) and the OTF mask. Once this mask has been established, then the moment method may make additional pixel selections.

- `drop` - If this is true (the default) then the moment axis is dropped from the output image. Otherwise, the output images have a moment axis of unit length and coordinate information that is the same as for the input image. This coordinate information may be totally meaningless for the moment images.

Finally, if you ask for a moment which requires the coordinate to be computed for each profile pixel (these are the intensity weighted mean coordinate [moment 1] and the intensity weighted dispersion of the coordinate [moment 2]), and the profile axis is not separable then there will be a performance loss. Examples of non-separable axes are RA and Dec. If the axis is separable (e.g. a spectral axis) there is no penalty. In the latter case, the vector of coordinates for one profile is the same as the vector for another profile, and it can be precomputed (once).
Note that this function has no "virtual" output file capability. All output files are written to disk. The output mask for these images is good (T) unless the moment method fails to generate a value (e.g. the total input pixel mask was all bad for the profile) in which case it will be bad (F).

Example

```
- im2 := im.moments(moments=[-1,1,2],axis=3,smoothaxes=[1,2,3],
                    smoothtypes="gauss gauss hann",smoothwidths=[5.0,5.0,3],
                    excludepix=[1e-3], smoothout='smooth')
```

In this example, standard moments (average intensity, weighted velocity and weighted velocity dispersion) are computed via the convolve (spatially convolved by gaussians and spectrally by a Hanning kernel) and clip method (we exclude any pixels with absolute value less than 0.001). The output file names are automatically created for us and the convolved image is saved. The returned image tool holds the first moment image.

Example

```
- im2 := im.moments(moments=[3],method='window',plotter='/glish',
                    nx=5,ny=5)
```

In this example, the median of each spectrum is computed, after pixel selection by the automatic window method. Each spectrum and the window are plotted (25 plots per page) on a PGPLOT X-window device. Because the plotting device is given and we did not give the noise level of the image, a histogram of the image is made and fit and we get to interactively do the Gaussian fit to the histogram. The output image is temporary and accessed via the returned Image tool.

---

image.momentsgui - Function

Package	**general**
Module	**images**
Tool	**image**

Compute moments from image via custom GUI interface

Synopsis
momentsgui(parent, widgetset)

Arguments

parent	in	Parent frame to embed GUI in Allowed: *Glish*Tk agent Default: Create stand alone
widgetset	in	The widgetset to use for the GUI Allowed: A widgetserver tool Default: dws

Returns
Agent or fail

Description

General
The moments function takes rather a lot of inputs, and the need for some of them is context dependent. This makes the command-line interface rather cumbersome to use. Therefore, this function provides a GUI interface to the `moments` function. This interface constrains your actions by disabling appropriate parts of the GUI depending upon what you choose to do. When you invoke this function, it will need to find the minimum and maximum values in your image, so function statistics is automatically run (this may cause you to see a progress meter).
Here is an example of the GUI.
The GUI is arranged into several 'rollups'. The first one, 'basic settings', contains the basic parameters you need to set. Other rollups are 'convolution', 'data selection' and 'plotting'. See moments for descriptions of the parameters contained in these.
If you don't provide a parent frame (the `parent` argument), the GUI is made in a standalone mode. This means it creates go, reset and dismiss buttons. If you provide the parent frame, you must also provide the buttons and act on them.
The primary mechanism for the context dependent disabling and enabling is the choice of method found in the 'basic settings' rollup. This presents all of the possible method choices via a multi-level menu (this is different from the moments function where you give a string containing up to three substrings, the combination of which dictates the method). Although this menu has several levels, it makes it unamibiguous as to what you are doing.
The methods are

- Basic method. This means the pixels are simply selected via a pixel inclusion or exclusion (or all) range. Optionally, you may apply the pixel range to the smoothed image. This produces a mask which is applied to the unsmoothed data.

- Window methods. The window method selects pixels in a coordinate range which surrounds the spectral feature of interest. The window can

be found from the smoothed or unsmoothed data, but the moments are always computed from the unsmoothed data.

  - Automatic methods. The automatic methods do not allow you to interact with the process.
    * Window, automatic, converging mean method. This is Albert Bosma's algorithm. An initial guess for a range of pixels surrounding a spectral feature is refined by widening until the mean of the pixels outside of the range converges (to the noise).
    * Window, automatic, fit gaussian method. The window is found by fitting a Gaussian to the spectral feature and by taking a coordinate range of $\pm 3\sigma$ around the centre of the line.
  - Interactive methods. The interactive methods allow you to interact with the process. They are very user intensive !
    * Window, interactive, direct method.
      You just specify the window directly with the cursor.
    * Window, interactive, fit gaussian method.
      Same as the automatic method except that you do the fit interactively.

- Fit methods. The fit methods fit a model to the spectrum. The moments are worked out from the fit, not the actual data.
  - Fit gaussian, automatic method. The fits are done without user interaction.
  - Fit gaussian, interactive method. The user interacts with the fitting process.

You will also find context dependent disabling and enabling when you select some of the convolution parameters; this interacts with parameters in "data selection". In particular, the idea of the convolution is to make a mask by selecting pixels from the convolved data. Therefore you must make a pixel inclusion or exclusion selection. Also, the median coordinate moment is only available with the basic method, so it is disabled for other methods.
The combination of a context dependent GUI and as much error checking as possible in the GUI, means you should not often have the underlying C++ application complain to you that you have not set some parameter.

Recovering the GUI
When you run this function, it returns an agent which is actually a reference to the underlying GUI. In addition, the GUI is left on the screen. You can explicitly dismiss it by pressing the Dismiss button or selecting Dismiss from the File menu. To recover the GUI you can do one of two things.

1. Run the command `momentsgui` again. This will return another reference of the GUI to you. However, it will reset some parts of the GUI because it is possible that the image changed since you last ran this function (the pixel values might have changed, it might even be a different `image file` attached to this `tool`).

   Thus

   ```
   - im := image('myimage')
   - g := im.momentsgui()              # Click Dismiss on GUI
   - g := im.momentsgui()              # Get GUI back with some reset
   ```

2. Apply the `gui` function to the reference returned when you ran the `momentsgui` function. This simply recovers the GUI with no resetting. If the image **has** changed, this will not reflect it.

   ```
   - im := image('myimage')
   - g := im.momentsgui()              # Click Dismiss on GUI
   - g.gui()                           # Get GUI back with no change
   ```

If you destroy the GUI (by selecting Done from the File menu) then you must run the `momentsgui` function again to regenerate it.

---

image.name - Function

Package	**general**
Module	**images**
Tool	**image**

Name of the image file this tool is attached to

Synopsis
name(strippath)

Arguments

strippath	in	Strip off the path before the actual file name ? Allowed:Bool Default: False

Returns
String or fail

Description

This function returns the name of the `image file` By default, this function returns the full absolute path of the `image file`. You can strip this path off if you wish with the `strippath` argument and just recover the `image file` name itself.

Example

```
- im := image('g1.app')
- im.name(strippath=F)
/DATA/ELARA_1/aips++/data/g1.app
- im.name(strippath=T)
g1.app
```

---

image.open - Function

Package	**general**
Module	**images**
Tool	**image**

Open a new image file with this image tool

Synopsis
open(infile)

Arguments

infile	in	image file name Allowed:String

Returns
T or fail

Description

Use this function when you are finished analyzing the current `image file` and want to attach to another one. This function detaches the `image tool` from the current `image file`, and reattaches it (opens) to the new `image file`.

The input image file may be in native AIPS++, FITS, or Miriad format. Look here for more information on foreign images.
You could, of course, also create a new `image tool` and associate that with the new `image file`, but this saves you the `tool` creation.

Example

```
- im:=image('myimage')              # create tool im and open image myimage
- im.open('anotherimage')           # attach tool im to image anotherimage
```

The `open` function first closes the old `image file`.

---

image.pixelvalue - Function

Package	**general**
Module	**images**
Tool	**image**

Get value of image and mask at specified pixel coordinate

Synopsis
pixelvalue(pixel)

Arguments

pixel	in	Pixel coordinate Allowed:Vector of integer Default: Reference pixel

Returns
Glish record, unset, or fail

Description

This function gets the value of the image and the mask at the specified pixel coordinate. The values are returned in a *Glish* record with fields 'value', 'mask' and 'pixel'. The value is returned as a quantity, the mask as a Bool (T is good). The 'pixel' field holds the actual pixel coordinate used.
If the specified pixel coordinate is off the image, unset is returned.

Excessive elements in `pixel` are silently discarded. Missing elements are given the (nearest integer) value of the reference pixel. This is reflected in the output record 'pixel' field.

Example

```
- im := imagemaketestimage();
- im.pixelvalue()
[mask=T, value=[value=2.50643158, unit=Jy/beam], pixel=[56 38] ]
-
- im.pixelvalue([0,0])
[i_am_unset=i_am_unset]
-
- im.pixelvalue([10])
[mask=T, value=[value=0.140122071, unit=Jy/beam], pixel=[10 38] ]
-
- im.pixelvalue([10,10,10])
[mask=T, value=[value=-0.452527285, unit=Jy/beam], pixel=[10 10] ]
```

---

image.putchunk - Function

Package	**general**
Module	**images**
Tool	**image**

Put pixels from a Glish array into a regular region of the image

Synopsis
putchunk(pixels, blc, inc, list, locking, replicate)

Arguments

pixels	in	Glish numeric array
blc	in	Bottom-Left-Corner (start) of location in image Allowed: Vector of integers Default: Start of image
inc	in	increment (stride) along axes Allowed: Vector of integers Default: 1
list	in	List bounding box to logger ? Allowed: T or F Default: F
locking	in	Unlock image after use ? Allowed: T or F Default: T
replicate	in	Replicate array through image Allowed: T or F Default: F

Returns
T or fail

Description

This function puts a *Glish* array into the `image file`. If there is a default `pixel mask` it is ignored in this process. It is the complement of the getchunk function. You can specify the `blc` and `inc` if desired. If they are unspecified, they default to the beginning of the image and an increment of one.
Any illegal `blc` values are set to unity. Any illegal `inc` values are set to unity. A fail will result if you attempt to put an array beyond the extent of the image (i.e., it is not truncated or decimated).
If there are fewer axes in the array than in the image, the array is assumed to have trailing axes of length unity. Thus, if you have a 2D array and want to put it in as the YZ plane rather than the XY plane, you must ensure that the shape of the array is [1,nx,ny].
However, the argument `replicate` can be used to replicate the array throughout the image (from the blc to the trc). For example, if you provide a 2D array to a 3D image, you can replicate it through the third axis by setting `replicate=T`. The replication is done from the specified `blc` to the end of the image. Use function putregion if you want to terminate the replication at a `trc` value.
The argument `locking` controls two things. If True, then after the function is called, the image is unlocked (so some other process can acquire a lock) and it is indicated that the image has changed (causes function view to redisplay the image if it is has been called). The reason for having this argument is that the

unlocking and updating processes are quite expensive. If you are repeatedly calling `putchunk` in a for loop, you would be advised to use this switch.
A related function is putregion which puts the pixels and masks into a more complex `region-of-interest`. Function `putchunk` is retained because it is faster and therefore preferable for repeated operation in loops if the `pixel mask` is not required.
See also the functions set and calc which can also change pixel values.

Example

We can clip all pixels to be <= 5 as follows.

```
- pix := im.getchunk()              # get all pixels into a Glish array
- pix[pix > 5] := 5                 # clip values to 5
- im.putchunk(pix)                  # put array back into image
```

The above example shows how you could clip an image to a value. If all the pixels didn't easily fit in memory, you would iterate through the image chunk by chunk to avoid exhausting virtual memory. Better would be to do this via LEL through function calc.
Suppose we wanted to set the fifth XY plane to 0.
We could do so as follows:

```
- im := image('myimage')
- imshape := im.shape()
- pix := array(0, imshape[1], imshape[2])
- im.putchunk(pix, blc=[1,1,5])
```

Suppose we wanted to set the first YZ plane to 0.

```
- pix := array(0, 1, imshape[2], imshape[3])
- im.putchunk(pix)
```

---

image.putregion - Function

Package	**general**
Module	**images**
Tool	**image**

Put pixels and mask into a region-of-interest of the image

Synopsis
putregion(pixels, pixelmask, region, list, usemask, locking, replicate)

Arguments

pixels	in	The pixel values Allowed: Glish numeric array Default: None
pixelmask	in	The pixel mask values Allowed: Glish Boolean array Default: None
region	in	The region of interest Allowed: Region tool Default: Whole image
list	in	List the bounding box and any mask creation to the logger Allowed: T or F Default: F
usemask	in	Honour the mask when putting pixels Allowed: T or F Default: T
locking	in	Unlock image after use ? Allowed: T or F Default: T
replicate	in	Replicate array through image Allowed: T or F Default: F

Returns
T or fail

Description

This function replaces data and/or `pixel mask` values in the image in the specified `region-of-interest`. The `pixels` and/or `pixelmask` arrays must be the shape of the bounding box, and the whole bounding box is replaced in the image. The `region-of-interest` is really only used to specify the bounding box. If the region extends beyond the image, it is truncated. If the `pixels` or `pixelmask` array shapes do not match the bounding box, an error will result. When you put a `pixel mask`, it either replaces the current default `pixel mask`, or is created. The `pixel mask` is put before the pixels.
The argument `usemask` is only relevant when you are putting pixel values and there is a `pixel mask` (meaning also the one you might have just put in place). If `usemask=T` then only pixels for which the mask is good (T) are altered. If `usemask=F` then all the pixels in the region are altered - the mask is ignored. The argument `replicate` can be used to replicate the array throughout the image (from the blc to the trc). For example, if you provide a 2D array to a

3D image, you can replicate it through the third axis by setting `replicate=T`. The replication is done in the specified `region`.
The argument `locking` controls two things. If True, then after the function is called, the image is unlocked (so some other process can acquire a lock) and it is indicated that the image has changed (causes function view to redisplay the image if it is has been called). The reason for having this argument is that the unlocking and updating processes are quite expensive. If you are repeatedly calling `putchunk` in a for loop, you would be advised to use this switch.
See the related functions putchunk, set and calc.

Example

Suppose that we have a 2-dimensional image. First we recover the pixel and `pixel mask` values from a polygonal region. Then we change the values in the array that are within the region to zero and replace the data.

```
- im := image('myimage')                    # Create image tool
- x := dq.quantity([3,6,9,6,5,5,3],'pix')     # X vector, in absolute pixels
- y := dq.quantity([3,4,7,9,7,5,5],'pix')     # Y vector, in absolute pixels
- r1 := drm.wpoly(x,y,csys=im.coordsys())    # Create polygonal world region
- local pixels, pixelmask
- im.getregion(pixels, pixelmask, r1, F)     # Recover pixels and mask
- pixels[pixelmask] := 0                     # Set pixels where mask is T to zero
- im.putregion(pixels=pixels, region=r1)     # Replace pixels only
```

---

image.rebin - Function

Package	**general**
Module	**images**
Tool	**image**

rebin the image by the specified factors

Synopsis
rebin(outfile, bin, region, mask, dropdeg, overwrite, async)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
bin	in	Binning factors for each axis Allowed:Vector of integers
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
dropdeg	in	Drop degenerate axes Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
Image tool or fail

Description

This function rebins the current image by the specified integer binning factors for each axis. The output pixel value is just the average of the input pixel values. The output pixel will be masked bad if there were no good input pixels. You cannot rebin a Stokes axis.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
Sometimes it is useful to drop axes of length one (degenerate axes). Use the `dropdeg` argument if you want to do this. It will discard the axes from the input image. Therefore the output shape and Coordinate System that you supply must be consistent with the input image after the degenerate axes are dropped.

Example

```
- im := imagemaketestimage();
- im2 := im.rebin(bin=[2,3]);
```

---

image.regrid - Function

Package	**general**
Module	**images**
Tool	**image**

regrid this image to the specified Coordinate System

Synopsis

regrid(outfile, shape, csys, axes, region, mask, method, decimate, replicate, doref, dropdeg, overwrite, force, async)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
shape	in	Shape of output image Allowed:Vector of integers Default: Input shape
csys	in	Coordinate System for output image Allowed:Coordsys tool Default: Input CoordinateSystem
axes	in	The output pixel axes to regrid Allowed:Vecor of integers Default: All
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
method	in	The interpolation method Allowed:String from 'nearest', 'linear', 'cubic' Default: 'linear'
decimate	in	Decimation factor for coordinate grid computation Allowed:Integer Default: 10
replicate	in	Replicate image rather than regrid ? Allowed:T or F Default: F
doref	in	Turn on reference frame changes Allowed:T or F Default: T
dropdeg	in	Drop degenerate axes Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
force	in	Force specified axes to be regridded Allowed:T or F Default: F
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
Image tool or fail

Description

This function regrids the current image onto a grid specified by the given Coordinate System. You can also specify the shape of the output image. The Coordinate System must be given via a Coordsys `tool`. It is optional; if not specified, the Coordinate System from the input image (i.e. the one to which you are applying the regrid function) is taken. The order of the coordinates and axes in the output image is always the same as the input image. It simply 'finds' the relevant coordinate in the supplied Coordinate System in order to figure out the regridding parameters. The supplied Coordinate System must have at least as many coordinates as are required to accomodate the axes you are regridding (e.g. if you regrid the first two axes, and these belong to a Direction Coordinate, you need one Direction Coordinate in the supplied Coordinate System). Coordinates pertaining to axes that are not being regridded are supplied from the input image, not the give Coordinate System.
Reference changes are handled (e.g. J2000 to B1950, LSR to TOPO). In general, the conversion machinery attempts to work out how sophisticated it needs to be (e.g. am I regridding LSR to LSR or LSR to TOPO). However, it errs on the side of conservatism so that it can be that the conversion machine requires more information than it actually needs. For full frame conversions, one needs to know things like location on earth (e.g. observatory), direction of observation, and time of observation.
If you get the above errors and you **are** doing a frame conversion, then that means you must insert some extra information into the Coordinate System of your image. Most likely it's the time (coordsys.setepoch) and location (coordsys.settelescope) that are missing. If you get these errors and you **know** that you are not specifying a frame change (e.g. regrid LSR to LSR) then try setting `doref=F`. This will (silently) bypass all possible frame conversions. Note that if you **are** requesting a frame conversion and you set `doref=F` you are doing a bad thing (and you will get no warnings).
If you regrid a plane holding a Direction Coordinate and the units are Jy/pixel then the output is scaled to conserve flux (roughly; just one scale factor at the reference pixel is computed).
A variety of interpolation schemes are provided (you need only specify the first three characters to `method`). The cubic interpolation is substantially slower than linear, and often the improvement is modest. By default you get linear interpolation.
You specify the shape of the output image (`shape`) and which output axes you want to regrid (`axes`). Note that a Stokes axis cannot be regridded (you will get a warning if you try and do this).
The `axes` argument cannot be used to discard axes from the output image; it can only be used to specify which **output** axes are going to be regridded and which are not. Any axis that you are not regridding must have the same output shape as the input image shape for that axis.
The `axes` argument can also be used to specify the order in which the **output**

axes are regridded. This may give you significant performance benefits. For example, imagine we are going to regrid a spectral-line cube of shape [512,512,1204] to shape [256,256,32]. If you specified `axes=[1,2,3]` then first, the Direction axes would be regridded for each of the 1024 pixels (and stored in a temporary image). Then each profile at each spatial location in the temporary image would be regridded to 32 pixels. You could speed this process up significantly by setting `axes=[3,1,2]`. In this case, first each profile would be regridded to 32 pixels, and then each plane of the 32 pixels would be regridded. Note that the order of `axes` does not affect the order of the `shape` argument. I.e. it should be given in the natural pixel axis order of the image `[256,256,32]` in both cases.
You can also specify a `region-of-interest` to be applied to the input image. If you do this, you need to be careful with the output shape for non-regridded axes (must match that of the region - use function boundingbox to find that out).
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
The argument `replicate` can be used to simply replicate pixels rather than regridding them. Normally (`replicate=F`), for every output pixel, its world coordinate is computed and the corresponding input pixel found (then a little interpolation grid is generated). If you set `replicate=T`, then what happens is that for every output axis, a vector of regularly sampled input pixels is generated (based on the ratio of the output and input axis shapes). So this just means the pixels get replicated (by whatever interpolation scheme you use) rather than regridded in world coordinate space. This process is much faster, but its not a true world coordinate based regrid.
As decribed above, when `replicate` is False, a coordinate is computed for each output pixel; this is an expensive operation. The argument `decimate` allows you to decimate the computation of that coordinate grid to a sparse grid, which is then filled in via fast interpolation. The default for `decimate` is 10. The number of pixels per axis in the sparse grid is the number of output pixels for that axis divided by the decimation factor. A factor of 10 does pretty well. You may find that for very non-linear coordinate systems (e.g. very close to the pole) that you have to reduce the decimation factor.
Sometimes it is useful to drop axes of length one (degenerate axes). Use the `dropdeg` argument if you want to do this. It will discard the axes from the input image. Therefore the output shape and Coordinate System that you supply must be consistent with the input image after the degenerate axes are dropped.
Argument `force` can be used to force all specified axes to be regridded, even if the algorithm determines that they don't need to be (because the input and output coordinate information is identical).
There is a useful function setreferencelocation that you can use to keep a specific world coordinate in the center of an image when regridding (see

example below).
The output `pixel mask` will be good (T) unless the regridding failed to find a value for that output pixel in which case it will be bad (F). For example, if the total input mask (default input `pixel mask` plus OTF mask) for all of the relevant input pixels were masked bad then the output pixel would be masked bad (F).
Multiple axis Coordinates limitation – Some cooordinates pertain to more than one axis. E.g. a Direction Coordinate holds longitude and latitude. A Linear Coordinate can also hold many axes. When you regrid *any* axis from a Coordinate which holds multiple axes, you must fully specify the coordinate information for all axes in that Coordinate in the Coordinate System that you provide. For example, you have a Linear Coordinate with two axes and you want to regrid axis one only. In the Coordinate System you provide, the coordinate information for axis two (not being regridded) must correctly be a copy from the input coordinate system (it won't be filled in for you).

Example

```
- imr := image('radio.image')
- imo := image('optical.image')
- cs := imo.coordsys();
- imrr := imr.regrid(outfile='radio.regridded', csys=cs,
                     shape=imo.shape())
- imrr.view()
- cs.done()
- imr.done()
- imo.done()
```

In this example, we regrid a radio image onto the grid of an optical image - this probably (if the optical FITS image was correctly labelled !!) will involve a projection change (optical images are usually TAN projection, radio usually SIN).

Example

```
- im := image('radio.image')
- cs := im.coordsys();
- cs.referencecode('dir')
J2000
- cs.setreferencecode(value='B1950', type='dir', adjust=T)
- im3 := im1.regrid(outfile='radio.regridded', csys=cs,
+                    shape=im2.shape())
```

In this example, we regrid a radio image from J2000 to B1950. This is accomplished by first recovering the Coordinate System into a Coordsys tool, manipulating the reference code with that `tool`, and then supplying the new Coordinate System to the regrid function.

Example

```
- im := imagemaketestimage('zz')
- cs := im.coordsys();
-
- p := im.shape() / 2.0
- p +:= 10;
- refval := im.toworld(pixel=p, format='n')          # Location of interest
- inc := cs.increment(quant=F)
- inc /:= 2.0;                                      # Halve increment
- cs.setincrement(value=inc)                        # Set increment
- shp := im.shape() * 2;                            # Double shape
- refpix := ((shp-1))/2.0 + 1;                      # New ref pix
- cs.setreferencelocation(pixel=refpix, world=refval)    # Center image on location of interest
-
- imr := im.regrid(csys=cs, shape=shp)        # Regrid
- cs.done()
- imagedones()
```

---

image.rotate - Function

Package	**general**
Module	**images**
Tool	**image**

rotate the direction coordinate axes attached to the image and regrid the image to the rotated Coordinate System

Synopsis

rotate(outfile, shape, pa, region, mask, method, decimate, replicate, overwrite, async)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
shape	in	Shape of output image Allowed:Vector of integers Default: Input shape
pa	in	Angle by which to rotate Allowed:Quantity, string, numeric Default: 0deg
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
method	in	The interpolation method Allowed:String from 'nearest', 'linear', 'cubic' Default: 'cubic'
decimate	in	Decimation factor for coordinate grid computation Allowed:Integer Default: 0
replicate	in	Replicate image rather than regrid ? Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
Image tool or fail

Description

This function rotates two axes of an image. These axes are either those associated with a Direction coordinate or with a Linear coordinate. The Direction coordinate takes precedence. If rotating a Linear coordinate, it must hold precisely two axes.
The method is that the Coordinate is rotated and then the input image is regridded to the rotated Coordinate System.
If the image brightness units are Jy/pixel then the output is scaled to conserve flux (roughly; just one scale factor at the reference pixel is computed).

A variety of interpolation schemes are provided (you need only specify the first three characters to `method`). The cubic interpolation is substantially slower than linear. By default you get cubic interpolation.

You can specify the shape of the output image (`shape`). However, all axis that are not regrided retain the same output shape as the input image shape for that axis. Only the direction coordinate axes are regridded.

You can also specify a `region-of-interest` to be applied to the input image. If you do this, you need to be careful with the output shape for non-regridded axes (must match that of the region - use function boundingbox to find that out).

If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.

The argument `replicate` can be used to simply replicate pixels rather than regridding them. Normally (`replicate=F`), for every output pixel, its world coordinate is computed and the corresponding input pixel found (then a little interpolation grid is generated). If you set `replicate=T`, then what happens is that for every output axis, a vector of regularly sampled input pixels is generated (based on the ratio of the output and input axis shapes). So this just means the pixels get replicated (by whatever interpolation scheme you use) rather than regridded in world coordinate space. This process is much faster, but its not a true world coordinate based regrid.

As decribed above, when `replicate` is False, a coordinate is computed for each output pixel; this is an expensive operation. The argument `decimate` allows you to decimate the computation of that coordinate grid to a sparse grid, which is then filled in via fast interpolation. The default for `decimate` is 0 (no decimation). The number of pixels per axis in the sparse grid is the number of output pixels for that axis divided by the decimation factor. A factor of 10 does pretty well. You may find that for very non-linear coordinate systems (e.g. very close to the pole) that you have to reduce the decimation factor.

The output `pixel mask` will be good (T) unless the regridding failed to find a value for that output pixel in which case it will be bad (F). For example, if the total input mask (default input `pixel mask` plus OTF mask) for all of the relevant input pixels were masked bad then the output pixel would be masked bad (F).

Example

```
- im := imagemaketestimage();
- im.view(axislabels=T);
- imr:=im.rotate(pa=45);
- imr.view(axislabels=T);
```

In this example, we rotate the direction coordinate axes (RA/Dec) of a test image by 45 degress and regrid the image onto the axes. The viewer displays the before and after images.

---

image.rename - Function

Package	**general**
Module	**images**
Tool	**image**

Rename the image file associated with this image tool

Synopsis
rename(name, overwrite)

Arguments

name	in	The new image file name Allowed:String
overwrite	in	Overwrite target file if it already exists Allowed:T or F Default: F

Returns
T or fail

Description

This function renames the `image file` associated with the `image tool`. If a file with name `name` already exists, you can overwrite it with the argument `overwrite`; otherwise a fail will result.

Example

```
- im := image('myimage')
- im.name(strippath=T)
myimage
- im.rename('newimage', overwrite=T)
- im.name(strippath=T)
newimage
```

image.replacemaskedpixels - Function

Package	**general**
Module	**images**
Tool	**image**

replace the values of pixels which are masked bad

Synopsis
replacemaskedpixels(pixels, region, mask, update, list)

Arguments

pixels	in	The new value(s) Allowed:Numeric scalar or LEL expression
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
update	in	Update mask as well ? Allowed:T or F Default: F
list	in	List the bounding box to the logger Allowed:T or F Default: F

Returns
T or fail

Description

This function (short-hand name `rmp`) replaces the values of all pixels whose total input mask (default input `pixel mask` and OTF mask) is bad (F) with the specified value.
If the argument `update` is F (the default), the actual `pixel mask` is left unchanged. That is, masked pixels remain masked. However, if you set `update=T` then the `pixel mask` will be updated so that the `pixel mask` will now be T (good) where the **total** input mask was F (bad).

See maskhandler for information on how to set the default `pixel mask`. There are a few ways in which you can specify what to replace the masked pixel values by.

- First, you can give the `pixels` argument a simple numeric scalar (e.g. `pixels=1.0`). Then, all masked values will be replaced by that value.

- Second, you can give a scalar LEL expression string (e.g. `pixels='min(myimage)'`). Then, all masked values will be replaced by the scalar that results from the expression. If the scalar expression is illegal (e.g. in the expression `pixels='min(myimage)'` there were no good pixels in *myimage*) then the value 0 is used for replacement.

- Third, you can give a LEL expression string which has the same shape as the `image file` you are applying the function to. For example, putting `pixels='myotherimage'` means replace all masked pixels in this `image file` with the equivalent pixel in the `image file` called *myotherimage*.

  Your expression might be quite complex, and you can think of it as producing another masked lattice. However, in the replace process, the mask of that expression lattice is ignored. Thus, only the mask of the `image file` you are replacing and the pixel values of the expression lattice are relevant.

  The expression must conform with the subimage formed by applying the `region-of-interest` to the image (i.e. that associated with this Image `tool`). If you use the `mask` argument as well, the `region-of-interest` is applied to it as well (see examples).

Example

```
- im := image('zz1')
- im.replacemaskedpixels(0.0)
- im.replacemaskedpixels('min(zz2)')
```

These examples replace all masked pixels by the specified scalar. In the second case, the scalar comes from a LEL expression operating on some other image *zz2*.

Example

```
- im := image('zz1')
- im.replacemaskedpixels(0.0, mask='zz2>0')
```

Let us say that *zz1* has no mask. By using the `mask` argument, we generate a transient mask which is T (good) when the pixel values are positive. This means that all non-positive values (when that mask is F [bad]) will be replaced with the value 0. If *zz1* did have a mask it would be applied as well as the transient mask (the masks would be logically ORed).

Example

```
- im := image('zz1')
- im2 := image('zz2')
- r := drm.quarter()
- im.replacemaskedpixels(0.0, region=r, mask='$im2>0')
```

The specified region takes one quarter of the image by area centered on the image center. The region is applied to the `mask` expression as well - this means that any images in the `mask` expression must conform with the *zz1* image. The replacement of the scalar is then done only within that region. Note that in the `mask` expression we have specified the image with the Image tool **im2** via the $ syntax (rather than referring to its disk file name *zz2*).

Example

```
- im := image('zz1')
- im.replacemaskedpixels('zz2+zz3')
```

In this example, the replacement values are taken from a LEL expression adding two other images together. The expression must conform with the image *zz1*.

Example

```
- im := image('zz1')
- r := drm.quarter()
- im.replacemaskedpixels('zz2[$r]+zz3[$r]', region=r)
```

In this example, the replacement values are taken from a LEL expression adding two other images together. Because expression must conform with the image *zz1* after application of the region, we must specify the region in the expression as well.

---

image.restoringbeam - Function

Package	**general**
Module	**images**
Tool	**image**

Get the restoringbeam

Synopsis
restoringbeam()

Returns
Glish record or fail

Description

This function (short-hand name `rb`) gets the restoring beam, if any. It is returned in a *Glish* record with fields 'major', 'minor' and 'postionangle'. Each of these fields contains a quantity. If there is no restoring beam, this function returds an empty record.
You can set the restoring beam with function setrestoringbeam.

Example

```
- im := image('hcn')
- im.restoringbeam()
[major=[value=35, unit=arcsec],
 minor=[value=13, unit=arcsec],
 positionangle=[value=80, unit=deg]]
```

---

image.sepconvolve - Function

Package	**general**
Module	**images**
Tool	**image**

Separable convolution

Synopsis
sepconvolve(outfile, axes, types, widths, scale, region, mask, overwrite, async)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
axes	in	Axes to convolve Allowed:Vector of integers Default: [1,2,...]
types	in	Type of convolution kernel Allowed:Vector of strings from 'boxcar', 'gaussian', and 'hanning' Default: 'gaussian'
widths	in	Convolution kernel widths Allowed:Vector of numeric, quantity or string
scale	in	Scale factor Allowed:Float Default: Autoscale
region	in	Region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
Image tool or fail

Description

This function (short-hand name `sc`) does Fourier-based convolution of an `image file` by a specified separable kernel.
If `outfile` is given, the image is written to the specified disk file. If `outfile` is unset, the Image `tool` is associated with a temporary image. This temporary image may be in memory or on disk, depending on its size. When you destroy the Image `tool` (with the done function) this temporary image is deleted.
You specify which axes of the image you wish to convolve, by what kernel of what width. The kernel types can be shortened to `'gauss'`, `'hann'` and `'box'`.
You specify the widths of the convolution kernels via the argument `widths`. The values can be specified as a vector of three different types.

- Quantity - for example `widths=dq.quantity("1arcsec 0.00001rad")`. Note that you can use pixel units, viz. `widths=dq.quantity("10pix 0.00001rad")` see below.
- String - for example `widths="1km 2arcsec"` (i.e. a string that the Quanta Constructor accepts).
- Numeric - for example `widths=[10,20]`. In this case, the units of the widths are assumed to be in pixels.

The interpretation of `widths` depends upon the kernel type.

- Gaussian - the specified width is the full-width at half-maximum.
- Boxcar (tophat) - the specified width is the full width.
- Hanning - The kernel is $z[i] = 0.25 * y[i-1] + 0.5 * y[i] + 0.25 * y[i+1]$. The width is always 3 pixels, regardless of what you give (but you still have to give it !).

The scaling of the output image is determined by the argument `scale`. If you leave it unset, then autoscaling will be invoked which means that the convolution kernels will all be normalized to have unit volume to as to conserve flux.
If you do not leave `scale` unset, then the convolution kernel will be scaled by this value (it has peak unity before application of this scale factor).
Masked pixels will be assigned the value 0.0 before convolution. The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too.
See also the other convolution functions convolve2d, convolve and hanning.

Example

```
- dowait := T
- im := image('xyv')
- im2 := im.sepconvolve(outfile='xyv.con', axes=[2,3], types="gauss box", widths=[10,2
```

---

image.sepconvolvegui - Function

Package	**general**
Module	**images**
Tool	**image**

Separable convolution via custom GUI interface

Synopsis
sepconvolvegui(parent, widgetset)

Arguments

parent	in	Parent frame to embed GUI in Allowed: *Glish*Tk agent Default: Create stand a alone
widgetset	in	The widgetset to use for the GUI Allowed: A widgetserver tool Default: dws

Returns
Agent or fail

Description

General
This function (short-hand name `scgui`) starts the custom GUI interface for the sepconvolve function. Here is an example of the GUI.
The GUI allows you to select which axes (listed by name) you wish to convolve. If you check the 'Convolve' button for any axis, then that indicates you wish to convolve that axis. The kernel types and widths widgets will then be active. You must fill in the widths for each selected axis. You can give the widths as simple numeric values (units of pixels assumed) or values with units like `10arcsec` or `20km/s`, `5pix` etc.
Popuphelp is on each button to tell you what they do.
If you don't provide a parent frame (the `parent` argument), the GUI is made in a standalone mode. This means it creates go, reset and dismiss buttons. If you provide the parent frame, you must also provide the buttons and act on them. Although this function returns a reference to the underlying subsequence for the custom GUI, you shouldn't need to do anything with it.

Recovering the GUI
When you run this function, it returns an agent which is actually a reference to the underlying GUI. In addition, the GUI is left on the screen. You can explicitly dismiss it by pressing the Dismiss button or selecting Dismiss from the File menu. To recover the GUI you can do one of two things.

1. Run the command `sepconvolvegui` again. This will return another reference of the GUI to you. However, it will reset some parts of the GUI

because it is possible that the image changed since you last ran this function (the pixel values might have changed, it might even be a different `image file` attached to this `tool`).

Thus

```
- im := image('myimage')
- g := im.sepconvolvegui()            # Click Dismiss on GUI
- g := im.sepconvolvegui()            # Get GUI back with some reset
```

2. Apply the `gui` function to the reference returned when you ran the `sepconvolvegui` function. This simply recovers the GUI with no resetting. If the image **has** changed, this will not reflect it.

```
- im := image('myimage')
- g := im.sepconvolvegui()            # Click Dismiss on GUI
- g.gui()                             # Get GUI back with no change
```

If you destroy the GUI (by selecting Done from the File menu) then you must run the `sepconvolvegui` function again to regenerate it.

---

image.set - Function

Package	**general**
Module	**images**
Tool	**image**

Set pixel and/or mask values with a scalar in a region-of-interest of the image

Synopsis

set(pixels, pixelmask, region, list)

Arguments

pixels	in	The pixel value Allowed:LEL scalar expression or numeric scalar Default: unset
pixelmask	in	The pixel mask value Allowed:T or F Default: unset
region	in	The region of interest Allowed:Region tool Default: Whole image
list	in	List the bounding box and any mask creation to the logger Allowed:T or F Default: F

Returns
T or fail

Description

This function replaces data and/or mask values into the image in the specified `region-of-interest`. You can think of it as a simplified version of the image calculator.
Unlike the calc function, you can only set a scalar value for all pixels in the specified `region-of-interest`. For example, it can be useful to set a whole image to one value, or a mask in a `region-of-interest` to one value. Although you could do that with the related functions putregion and putchunk, you would have to make an array of the shape of the image and if that is large, it could be resource expensive.
The value for the pixels is specified with the `pixels` argument. It can be given as either a Lattice Expression Language (or LEL) expression string or a simple numeric scalar. See note 223 for a detailed description of the LEL expression syntax. If you give a LEL expression it must be a scalar expression.
Note that any default mask is *ignored* by this function when you set pixel values. This is different from calc where the extant mask is honoured.
The value for the pixel mask is specified with the `pixelmask` argument (`T, F, unset`). If it's `unset` then the mask is not changed.
If you specify `pixelmask=` T or F, then the mask that is affected is the current default mask (see maskhandler). If there is no mask, a mask is created for you and made the default mask.

Example

```
- im := imagefromshape('xx', [10,20])
- r1 := drm.box([2,2],[6,8])         # Make a box region
- im.set(pixels=1.0)                 # Set all pixels to 1
- im.set(pixels='1.0', region=r1)    # Set all pixels to 1 in the region
- im.set(pixels='min(yy)')           # Set all pixels to minimum of image yy
- im.set(pixels='min($im)', region=r1) # Set pixels in region to minimum of image xx
- im.set(pixelmask=T)                       # Set mask to all T
- im.set(pixels=0, pixelmask=F, region=r1)  #Set pixels and mask in region
```

---

image.setbrightnessunit - Function

Package	**general**
Module	**images**
Tool	**image**

Set the image brightness unit

Synopsis
setbrightnessunit(unit)

Arguments

unit	in	New brightness unit Allowed:String

Returns
T or fail

Description

This function (short-hand name `sbu`) sets the image brightness unit. You can get the brightness unit with function brightnessunit.

Example

```
- im := imagefromshape(shape=[10,10])
- im.setbrightnessunit('km')
- im.brightnessunit()
km
```

---

image.setcoordsys - Function

Package	**general**
Module	**images**
Tool	**image**

Set new Coordinate System

Synopsis
setcoordsys(csys)

Arguments

csys	in	Coordinate System Allowed:Coordsys tool

Returns
T or fail

Description

This function replaces the Coordinate System in the image. Coordinate System are stored in and manipulated with the Coordsys `tool`. The Coordinate System can be recovered from an image via the coordsys function. Note that changing the Coordsys `tool` has no effect on the original image, until it is replaced with this function.

Example

```
- im := imagefromshape(shape=[10,20])  # Make image
- cs := im.coordsys();                 # Recover Coordinate System
- incr := cs.increment('n');             # Get increment as numeric vector
- cs.setincrement(value=2*incr);       # Set new increment in Coordinate System
- im.setcoordsys(cs);                  # Set new Coordinate System in image
```

---

image.sethistory - Function

Package	**general**
Module	**images**
Tool	**image**

Set the history for an image

Synopsis
sethistory(history)

Arguments

history	in	New history Allowed:String or vector of strings Default: None

Returns
T or fail

Description

An AIPS++ `image file` can accumulate history information from an input FITS file or by you writing something into it explicitly with this function. Each element of the input vector is one line of history. The new history is appended to the old.
Presently, AIPS++ functions do not automatically add history records. You can recover the old history with function history.

Example

```
- im := image('myfile')
- h := im.history()
- im.sethistory("I like fish")       # Adds three lines, 'I', 'like' and 'fish'
```

---

image.setmiscinfo - Function

Package	**general**
Module	**images**
Tool	**image**

Set the miscellaneous information record for an image

Synopsis
setmiscinfo(info)

Arguments

info	in	Miscellaneous REPLACEMENT header Allowed:Record

Returns
T or fail

Description

An AIPS++ `image file` can accumulate various miscellaneous information during its lifetime; it is stored in a *Glish* record called the `miscinfo` record. For example, the FITS reader (imagefromfits) puts header keywords it doesn't otherwise use into the `miscinfo` record. The `miscinfo` record is not guaranteed to have any entries, so it's up to you to check for any fields that you require.
This function sets the `miscinfo` record of the `image file`. Note that this function *replaces* the record, it doesn't add to it, so if you want to augment the existing record, you should first capture it with the miscinfo) function, add to the record, and then put it back. The FITS writer will attempt to write all the fields in the `miscinfo` record to FITS file argument. It can do so for scalars and 1-dimensional arrays. Records will be omitted, and multi-dimensional arrays will be flattened into 1-dimensional arrays.

Example

```
- im := image('myfile')
- x := something(im)      # calculate something interesting from im
- info := im.miscinfo()   # capture the miscinfo record
- info.mycalc := x        # add our result to it
- im.setmiscinfo(info)    # put it back into the image
```

---

image.shape - Function

Package	**general**
Module	**images**
Tool	**image**

Length of each axis in the image

Synopsis
shape()

Returns
Vector of integers or fail

Description

The shape of an image is a vector holding the length of each axis of the image. Although this information is also available in the summary function, it is so useful that it has been separated out into a separate function.

Example

```
- imshape := im.shape()
- npixels := prod(imshape)    # imshape[1]*imshape[2]*...*imshape[n]
```

---

image.setrestoringbeam - Function

Package	**general**
Module	**images**
Tool	**image**

Set the restoringbeam

Synopsis
setrestoringbeam(major, minor, pa, beam, delete, log)

Arguments

major	in	Major axis FWHM Allowed:Quantity or float Default: Unset
minor	in	Minor axis FWHM Allowed:Quantity or float Default: Unset
pa	in	Position angle Allowed:Quantity or float Default: Unset
beam	in	The complete restoring beam Allowed:Record Default: Unset
delete	in	Delete the restoring beam Allowed:T or F Default: F
log	in	Write new beam values to the logger Allowed:T or F Default: F

Returns
T or fail

Description

This function (short-hand name `srb`) sets the restoring beam. Do not do this unless it is a correct thing to do !
You may supply the beam in one of two ways.
First, you can use the argument `beam` which you must assign to a *Glish* record containing fields 'major', 'minor' and 'postionangle'. Each of these fields contains a quantity. This record is in the same format as one returned by function restoringbeam. If you leave `beam` unset, no action with it is taken. If the function argument `beam` is used, the arguments `major, minor, & pa` are ignored.
Second, you can use the arguments `major`, `minor` and `pa`. Only the ones that you assign are used. Each argument should be assigned either a quantity or a float (units are implicitly those of the current beam - or if none, arcsec for the axes and degrees for the position angle).
You may also delete the restoring beam by setting `delete=T` (all other arguments are then ignored).

Example

```
- im := image('hcn')
- rb := im.restoringbeam()
- rb
[major=[value=35, unit=arcsec],
 minor=[value=13, unit=arcsec],
 positionangle=[value=80, unit=deg]]
-
- rb.minor.value := 12.5
- im.setrestoringbeam(beam=rb)
- im.restoringbeam()
[major=[value=35, unit=arcsec],
 minor=[value=12.5, unit=arcsec],
 positionangle=[value=80, unit=deg]]
-
- im.setrestoringbeam(major=36)
- im.restoringbeam()
[major=[value=36, unit=arcsec],
 minor=[value=12.5, unit=arcsec],
 positionangle=[value=80, unit=deg]]
```

---

image.statistics - Function

Package	**general**
Module	**images**
Tool	**image**

Compute statistics from the image

Synopsis

statistics(statsout, axes, region, mask, plotstats, includepix, excludepix, plotter, nx, ny, list, force, disk, robust, verbose, async)

Arguments

statsout	out	*Glish* record containing all statistics Allowed: *Glish* record Default: No retrieval into a *Glish* record
axes	in	List of axes to evaluate statistics over Allowed: Vector of integers Default: All axes
region	in	Region of interest Allowed: Region tool Default: Whole image
mask	in	OTF mask Allowed: Boolean LEL expression or mask region Default: None
plotstats	in	List of statistics to plot Allowed: String containing any combination of 'npts', 'sum', 'flux', 'sumsq', 'min', 'max', 'mean', 'sigma' (or 'stddev'), 'rms', 'median', 'medabsdevmed', 'quartile' Default: 'mean sigma'
includepix	in	Range of pixel values to include Allowed: Vector of 1 or 2 Floats Default: Include all pixels
excludepix	in	Range of pixel values to exclude Allowed: Vector of 1 of 2 Floats Default: Exclude no pixels
plotter	in	The PGPLOT device name to make plots on Allowed: Any valid PGPLOT device Default: No plotting
nx	in	The number of subplots per page in the x direction Allowed: Any positive integer Default: 1
ny	in	The number of subplots per page in the y direction Allowed: Any positive integer Default: 1
list	in	If T then list the statistics as well Allowed: Bool Default: T
force	in	If T then force the stored statistical accumulations to be regenerated Allowed: Bool Default: F
disk	in	If T then force the storage image to disk Allowed: Bool Default: F
robust	in	If T then compute robust statistics as well Allowed: Bool Default: F
verbose	in	If T then print useful messages to the logger Allowed: Bool Default: T
async	in	Run asynchronously? Allowed: Bool Default: !dowait - but always F if plotting

Returns
T or fail

Description

This function (short-hand name is `stats`) computes statistics from the pixel values in the image. You can then plot them, list them and retrieve them (into a *Glish* record) for further analysis.
The chunk of the image over which you evaluate the statistics is specified by an array of axis numbers (argument `axes`). For example, consider a 3-dimensional image for which you specify `axes=[1,3]`. The statistics would be computed for each XZ (axes 1 and 3) plane in the image. You could then examine those statistics as a function of the Y (axis 2) axis. Or perhaps you set `axes=[3]`, whereupon you could examine the statistics for each Z (axis 3) profile as a function of X and Y location in the image.
You have control over how the statistics are presented; whether they are listed, or plotted, or retrieved into *Glish* records or all of the above.
A variety of statistics are offered for plotting and you can plot any combination of them simultaneously (argument `plotstats`).
If you retrieve the statistics into a *Glish* record (argument `statsout`) each statistic is stored in an array in one named field in the *Glish* record. The shape of that array is that of the axes which you did **not** evaluate the statistics over. For example, in the second example above, we set `axes=[3]` and asked for statistics as a function of the remaining axes, in this case, the X and Y (axes 1 and 2) axes. The shape of each statistics array is then [nx,ny].
The names of the fields in this record are the same as the names of the statistics that you can plot:

- `npts` - the number of unmasked points used
- `sum` - the sum of the pixel values: $\sum I_i$
- `flux` - the integrated flux density if the beam is defined and the brightness units are $Jy/beam$
- `sumsq` - the sum of the squares of the pixel values: $\sum I_i^2$
- `mean` - the mean of pixel values: $\bar{I} = \sum I_i/n$
- `sigma` - the standard deviation about the mean: $\sigma^2 = (\sum I_i - \bar{I})^2/(n-1)$
- `rms` - the root mean square: $\sqrt{\sum I_i^2/n}$
- `min` - minimum pixel value
- `max` - the maximum pixel value
- `median` - the median pixel value (if `robust=T`)

- `medabsdevmed` - the median of the absolute deviations from the median (if `robust=T`)
- `quartile` - the inter-quartile range (if `robust=T`). Find the points which are 25% largest and 75% largest (the median is 50% largest), find their difference and divide that difference by 2.
- `blc` - the absolute pixel coordinate of the bottom left corner of the bounding box
- `blcf` - the formatted absolute world coordinate of the bottom left corner of the bounding box
- `trc` - the absolute pixel coordinate of the top right corner of the bounding box
- `trcf` - the formatted absolute world coordinate of the top right corner of the bounding box
- `minpos` - absolute pixel coordinate of minimum pixel value
- `maxpos` - absolute pixel coordinate of maximum pixel value
- `minposf` - formatted string of the world coordinate of the minimum pixel value
- `maxposf` - formatted string of the world coordinate of the maximum pixel value

the last four fields only appear if you evaluate the statistics over all axes in the image. As an example, if your output *Glish* record is called `'mystats'`, then you could access the 'mean' field via `print mystats.mean`.
If there are no good points (e.g. all pixels are masked bad in the region), then the length of these fields will be 0 (e.g. `length(rec.npts)==0`).
If you retrieve statistics all of the available statistics are retrieved into the *Glish* record regardless of what you might have asked to plot. You can of course also plot the statistics by accessing the arrays in the *Glish* record, rather than using the built in plotting capability.
You also have no control over which statistics are listed to the logger, you always get the same selection. You can choose to list the statistics or not (argument `list`).
As well as the simple (and faster to calculate) statistics like means and sums, you can also compute some robust statistics. Currently these are the median, median absolute deviations from the median and the inter-quartile range. Because these are computationally expensive, they are only computed if you directly ask for them. You can do this in two ways. First by requesting a robust statistic in the list of statistics to plot. Second by setting `robust=T` (it defaults to F). For example, it may be that you are not making any plots but

wish to recover the robust statistics into the output record. This is why this extra argument is needed.
Note that if the axes are set to all of the axes in the image (which is the default), no plotting is available as there would be just one point per statistic.
You have control over which pixels are included in the statistical sums via the `includepix` and `excludepix` arguments. These vectors specify a range of pixel values for which pixels are either included or excluded. They are mutually exclusive; you can specify one or the other, but not both. If you only give one value for either of these, say `includepix=b`, then this is interpreted as `includepix=[-abs(b),abs(b)]`.
The plotting is done directly on a PGPLOT plotting device. The syntax is `plotter=name/type`. For example `plotter='plot1.ps/ps'` (disk postscript file) or `plotter='plot/glish'` (*Glish* pgplotter).
This function generates a 'storage' lattice, into which the statistics are written. It is only regenerated when necessary. For example, if you run the function twice with identical arguments, the statistics will be directly retrieved from the storage lattice the second time. However, you can force regeneration of the storage image if you set `force=T`. The storage medium is either in memory or on disk, depending upon its size. You can force it to disk if you set `disk=T`, otherwise it decides for itself.

Example

```
- im.statistics()
```

In this example, we ask to see statistics evaluated over the entire image. No plotting is available under these conditions.

Example

```
- im.statistics(statsout=mystats,axes=[2,3],plotstats='sigma, rms',
+               includepix=[0,100],plotter='/glish',list=F)
```

In this example, let us assume the image has 3 dimensions. We plot the standard deviation about the mean and the rms of YZ (axes 2 and 3) planes for pixels with values in the range 0 to 100 as a function of X-axis location on the standard PGPLOT X-windows device. The statistics are not listed to the logger. Because we are making plots, the tool function is automatically run synchronously so we can directly recover the the *Glish* record containing the statistics arrays.

---

image.twopointcorrelation - Function

Package	**general**
Module	**images**
Tool	**image**

Compute two point correlation function from the image

Synopsis
twopointcorrelation(outfile, region, mask, axes, method, overwrite)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
axes	in	The pixel axes to compute structure function over Allowed:Vector of integers Default: Sky or first two
method	in	The method of computation Allowed:String from 'structurefunction' Default: structurefunction
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
T or fail

Description

This function (short-hand name is `tpc`) computes two-point auto-correlation functions from an image.
By default, the auto-correlation function is computed for the Sky axes. If there is no sky in the image, then the first two axes are used. Otherwise you

can specify which axes the auto-correlation function lags are computed over with the `axes` argument (must be of length 2).
Presently, only the Structure Function is implemented. This is defined as :

$$S(lx, ly) =< (data(i, j) - data(i + lx, j + ly))^2 >$$

where lx, ly are integer lags in the x (first axis) and y (second axis) directions. The ensemble average is over the all values at the same lag pair. This process is extremely compute intensive and so you may have to be patient.
In an auto-correlation function image there are some symmetries. The first and third quadrants are symmetric, and the second and fourth are symmetric. So in principle, all the information is in the top or bottom half of the image. We just write it all out to look nice. The long lags don't have a lot of contributing values of course.

Example

```
- im := imagemaketestimage();        # Output image is virtual
- im2 := im.twopointcorrelation()    # Output image is virtual
```

---

image.subimage - Function

Package	**general**
Module	**images**
Tool	**image**

Create a (sub)image from a region of the image

Synopsis
subimage(outfile, region, mask, dropdeg, overwrite, list)

Arguments

outfile	in	Output image file name Allowed:String Default: unset
region	in	Region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
dropdeg	in	Drop degenerate axes Allowed:T or F Default: F
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
list	in	List informative messages to the logger Allowed:T or F Default: T

Returns
Image tool or fail

Description

This function (short-hand name `subim`) copies all or part of the image to another (there is also a subimage constructor).
If `outfile` is given, the subimage is written to the specified disk file. If `outfile` is unset, the Image `tool` actually references the input image file (i.e. that associated with the Image `tool` to which you are applying this function). So if you deleted the input image disk file, it would render this `tool` useless. When you destroy this `tool` (with the done function) the reference connection is broken.
Sometimes it is useful to drop axes of length one (degenerate axes). Use the `dropdeg` argument if you want to do this.
The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask. Any other input `pixel masks` will not be copied. Use function maskhandler if you need to copy other masks too.

Example

```
- im := image('myfile')
- im2 := im.subimage()                                    # a complete copy
```

```
- r1 := drm.box([10,10],[30,40],[5,5])                    # A strided pixel box reg
- im3 := im.subimage(outfile='/tmp/foo', region=r1)       # Explicitly named subima
```

---

image.summary - Function

Package	**general**
Module	**images**
Tool	**image**

Summarize basic information about the image

Synopsis
summary(header, doppler, list, pixelorder)

Returns
T or fail

Arguments

header	out	Header information Default: empty
doppler	in	If there is a spectral axis, list velocity too, with this doppler definition Allowed:String Default: RADIO
list	in	List the summary to the logger Allowed:T or F Default: T
pixelorder	in	List axis descriptors in pixel or world axis order Allowed:T or F Default: T

Description

This function summarizes miscellaneous information such as shape, Coordinate System, restoring beams, and masks.
If called without any arguments, this function displays a summary of the image header to the logger only; where appropriate, values will be formatted nicely (e.g. HH:MM:SS.SS for the reference value of RA axes).

For spectral axes, the information is listed as a velocity as well as a frequency. The argument `doppler` allows you to specify what velocity doppler convention it is listed in. You can choose from `radio, optical` and `true`. Alternative names are `z` for `optical`, and `beta` or `relativistic` for `true`. The default is `radio`. The definitions are

- radio: $1 - F$
- optical: $-1 + 1/F$
- true: $(1 - F^2)/(1 + F^2)$

where $F = \nu/\nu_0$ and ν_0 is the rest frequency. If the rest frequency has not been set in your image, you can set it via a Coordsys `tool` with the function setrestfrequency.
If the `header` argument is supplied, that argument will be a *Glish* record with the following fields filled in:

ndim Dimension of the image.

shape Length of each axis in the image.

tileshape Shape of the chunk which is most efficient for I/O.

axisnames Name of each axis.

refpix Reference pixel for each axis (1 relative)

refval Reference value for each axis.

incr Increment for each axis.

axisunits Unit name for each axis.

unit Brightness units for the pixels.

hasmask True if the image has a mask.

defaultmask The name of the mask which is applied by default.

masks The names of all the masks stored in this image.

restoringbeam The restoring beam if present.

imagetype The image type.

If you set `list=F`, then the summary will not be written to the logger. However, in addition to the output record, the return value of the function will be a vector strings containing the formatted output that would normally have gone to the logger. If `list=T` the return value is always T or a fail.

Example

```
- im := image('myim1')
- im.summary()                 # summarize to logging only

Image name       : test
Image mask       : mask0 [mask1, mask2]
Image units      : JY/BEAM

Direction reference : J2000
Spectral  reference : LSRK
Velocity  type      : RADIO
Rest frequency      : 1.42041e+09 Hz

Telescope        : ATCA
Observer         : NEBK
Date observation : 1997/02/121/03:03:26

Axis Type      Name             Proj Shape Tile   Coord value at pixel    Coord incr Ur
-------------------------------------------------------------------------------------
1    Direction Right Ascension   SIN  512   10  00:00:00.000     6.00 -6.000000e+01 ar
2    Direction Declination       SIN  512   20 +00.00.00.000    11.00  6.000000e+01 ar
3    Spectral  Frequency             1024   30  1.415000e+09    16.00  1.000000e+03 Hz
               Velocity                        1.140944e+03    16.00 -2.110611e-01 km/

- im.summary(header,list=F)      # store header in Glish record
- if (header.ndim == 3) ...      # program using header values
```

Note that the Spectral Coordinate is listed twice; once with frequency and once with velocity units. You can distinguish this double listing because the shape of the axis is only listed once (in this case on the Frequency line). Note also that this image has three masks present. The first one is the default which means that it is applied by applications. Any of the other masks, listed in square brackets, could be made the default mask by application of the maskhandler function. If there was no default mask (effectively that means an all good mask would be applied), all the mask names would be listed in square brackets.

---

image.tofits - Function

Package	**general**
Module	**images**
Tool	**image**

Convert the image to a FITS file

Synopsis
tofits(outfile, velocity, optical, bitpix, minpix, maxpix, region, mask, overwrite, dropdeg, deglast, async)

Arguments

outfile	in	FITS file name Allowed:String Default: Input name + '.fits'
velocity	in	prefer velocity (rather than frequency) as primary spectral axis? Allowed:Bool Default: F
optical	in	use the optical (rather than radio) velocity convention? Allowed:Bool Default: T
bitpix	in	Bits per pixel Allowed:-32 (floating point) or 16 (integer) Default: -32
minpix	in	Minimum pixel value for BITPIX=16 Allowed:Float Default: Autoscale
maxpix	in	Maximum pixel value for BITPIX=16 Allowed:Float Default: Autoscale
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F
dropdeg	in	Drop degenerate axes ? Allowed:T or F Default: F
deglast	in	Put degenerate axes last in header ? Allowed:T or F Default: F
async	in	Run asynchronously? Allowed:Bool Default: !dowait

Returns
T or fail

Description

This function converts the image into a FITS file.

If the image has a rest frequency associated with it, it will always write velocity information into the FITS file. By default the frequency information will be primary as it is better defined than velocity information. If you select `velocity=T` then by default the velocity is written in the optical convention, but if `optical=F` it will use the radio convention instead.
The FITS definition demands equal increment pixels. Therefore, if you write velocity information as primary and use the optical velocity definition, the increment is computed at the spectral reference pixel. If the bandwidth is large, this may incur non-negligible coordinate calculation errors far from the reference pixel.
By default the image is written as a floating point FITS file (`bitpix= -32`). Under rare circumstances you might want to save space and write it as scaled 16 bit integers (`bitpix = 16`). You can have `tofits` calculate the scaling factors by unsetting `minpix` and `maxpix` (the default). If you set `minpix` and `maxpix`, values outside of that range will be truncated. This can be useful if all of the FITS images dynamic range is being used by a few high or low values and you are not interested in preserving those values exactly. Besides the factor of two space savings you get by using 16 instead of 32 bits, integer images usually also compress well (for example, with the standard GNU software facility `gzip`).
If the specified `region-of-interest` extends beyond the image, it is truncated.
The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask.
Sometimes it is useful to drop axes of length one (degenerate axes) because not all FITS readers can handle them. Use the `dropdeg` argument if you want to do this.

Example

```
- ok := im.tofits('MYFILE.FITS')             # write FITS image file
- ok := im.tofits('MYFILE2.FITS', bitpix=16) # Write as scaled 16 bit integers
```

---

image.toascii - Function

Package	**general**
Module	**images**
Tool	**image**

Convert the image to an ascii file

Synopsis
toascii(outfile, region, mask, sep, format, maskvalue, overwrite)

Arguments

outfile	in	Ascii file name Allowed:String Default: Input name + '.ascii'
region	in	The region of interest Allowed:Region tool Default: Whole image
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
sep	in	Separator of data in ascii file Allowed:String Default: ' '
format	in	Format of data in ascii file Allowed:String Default: %e
maskvalue	in	Value to replace masked pixels by Allowed:Float or unset Default: No change
overwrite	in	Overwrite (unprompted) pre-existing output file ? Allowed:T or F Default: F

Returns
T or fail

Description

This function converts the image into an ascii file. The format is one image row per line (see imagefromascii).
The output mask is the combination (logical OR) of the default input `pixel mask` (if any) and the OTF mask. Because the mask is not transferred to the ascii file, you must specify what data value to use if a pixel is masked. By default, the underlying data value in the image is used. But this could be anything (and often it's a NaN), so you could set, say, `maskvalue=-10000` as a magic value.

Example

```
- im := imagemaketestimage()
- ok := im.toascii('myfile.ascii')
```

---

image.type - Function

Package	**general**
Module	**images**
Tool	**image**

Return the type of this tool

Synopsis
type()

Returns
String or fail

Description

This function returns the string 'image'. It is used so that in a script, you can make sure this *Glish* variable is an Image `tool`.

---

image.topixel - Function

Package	**general**
Module	**images**
Tool	**image**

Convert from world to pixel coordinate

Synopsis
topixel(value)

Arguments

value	in	Absolute world coordinate Allowed:Numeric vector, vector of quantities or record Default: Reference value

Returns
Vector of float or fail

Description

This function converts from absolute world to pixel coordinate (1-rel). The world coordinate can be provided in many formats (numeric, string, quantum etc.) via the argument `value`. These match the output formats of function toworld.
This function is just a wrapper for the Coordsys `tool` function topixel so see the documentation there for a description and more examples.

Example

```
- im := imagemaketestimage();
- w := im.toworld([10,10], 'n')        # Numeric vector
- im.topixel(w)
[10 10]
- w := im.toworld([10,10], 'm')        # Record of measures
- im.topixel(w)
[10 10]
```

Convert to a pixel coordinate to world as floats and then back to pixel. Do the same with the world coordinate formatted as measures instead.

---

image.toworld - Function

Package	**general**
Module	**images**
Tool	**image**

Convert from pixel to world coordinate

Synopsis
toworld(value, format)

Arguments

value	in	Absolute pixel coordinate Allowed:Numeric vector Default: Reference pixel
format	in	What type of formatting ? Allowed:String from combination of 'n' (numeric), 'q' (quantity), 'm' (measure), 's' (string) Default: 'n'

Returns
Vector float, vector quantity, record, string or fail

Description

This function converts between absolute pixel coordinate (1-rel) and world (physical coordinate).
This function is just a wrapper for the Coordsys `tool` function toworld so see the documentation there for a description of the arguments and more examples.

Example

```
- im := image('hcn')
- w := cs.toworld([10,10], 'n')
- w := cs.toworld([10,10], 'nmq')
```

Convert to a vector of floats and then to a record holding a vector of floats, a vector of quantities and a subrecord of measures.

---

image.unlock - Function

Package **general**
Module **images**
Tool **image**

Release any lock on the image

Synopsis
unlock()

Description

This function releases any lock set on the `image file` (and also flushes any outstanding I/O to disk). It is not of general user interest. It can be useful in scripts when a file is being shared between more than one process. See also functions lock and haslock.

Example

```
- im := imagefromarray('xx', array(0,10,20))
- im.unlock()
```

This releases the write lock on the `image file`. Now some other process can gain immediate access to the `image file`.

---

image.view - Function

Package	**general**
Module	**images**
Tool	**image**

Display the current image with the viewer

Synopsis

view(parent, raster, contour, vector, marker, region, mask, model, adjust, axislabels, includepix, order, activatebreak, hasdimiss, widgetset)

Arguments

parent	in	Parent frame to embed display in Allowed:Glish/Tk frame agent Default: F
raster	in	Display as a raster Allowed:unset, T or F Default: Last value (T if first)
contour	in	Display as contours Allowed:unset, T or F Default: Last value (F is first)
vector	in	Display as vectors Allowed:unset, T or F Default: Last value (F is first)
marker	in	Display as marker Allowed:unset, T or F Default: Last value (F is first)
region	in	Region of interest Allowed:Region tool Default: Last value (whole image if first)
mask	in	OTF mask Allowed:Boolean LEL expression or mask region Default: None
model	in	Componentlist Allowed:Componentlist tool Default: None
adjust	in	Start adjustment GUI ? Allowed:T or F Default: T
axislabels	in	Display axis labels Allowed:unset, T, F Default: Last value (F if first)
includepix	in	Pixel range Allowed:unset or Vector of float (length 1 or 2) Default: Full range
order	in	Order to display axes in Allowed:unset or vector integer Default: Last value (natural if first)
activatebreak	in	Activate the break button Allowed:unset, T, F Default: unset
hasdimiss	in	Add Dismiss button Allowed:T, F Default: T
widgetset	in	The widgetset to use for the GUI Allowed:A widgetserver tool Default: ddlws

Returns
T or fail

Description

- General

  This function displays the image with the viewer. The viewer is built with the Display Library.

  This function displays the image as one or more of a raster display, contour map, vector map or marker map. The default behaviour is to display first as a raster. If `raster`, `contour`, `vector` or `marker` is left unset, then the behaviour is to leave the raster or contour or vector display, respectively, as they were last time. You can set these arguments to T or F explicitly to get what you want otherwise.

  You can embed the display panel it creates in an existing frame if you wish (`parent`). You can optionally start the adjustment control (`adjust=T`) GUI (one for each of the raster or contour displays) which gives you control over things like axis labels, contour levels etc.

  You can specify the `region-of-interest` you want to see with the `region` argument. By default, you see the whole image. If you call this function again without specifying the `region` argument, then you get whatever `region-of-interest` you last displayed. You must explicitly over-ride `region` to change the displayed `region-of-interest`.

  You can specify the pixel range used to map to the display colors via the `includepix` argument. If unset (the default) the full range is taken. Otherwise you can specify 1 or 2 arguments. If you only give one value for either of these vectors, say `includepix=[b]`, then this is interpreted as `includepix=[-abs(b),abs(b)]`. If you give two values, then these are the data minimum and maximum

  The argument `order` allows you to control the initial display order of the axes in the image. For example, if your image has axes RA,DEC,Stokes,Frequency and you set `order=[4,1,2]` then the x-axis will be Frequency, the y-axis RA and the z-axis (movie axis) DEC. You can specify at most three axes. The hidden axes will sort themselves out. The first time, by default you get the natural order. Thereafter, if `orer` is unset you get the order you last specified.

  The argument `hasdismiss` controls whether a Dismiss button is added to the GUI or not. It always has a Done button. This is argument is not for general user use.

  If you dismiss the display panel (press dismiss button) call the `view` function again to bring it back. If you destroy the display panel (press

done button) it will be recreated next time you run the `view` function (takes longer than just redisplaying the GUI of course).

An example display from this function as well as a full description of most of it's GUI elements is shown here.

In this case, the 2D image is shown as a raster image with the default 'Rainbow 1' transfer function. The axis labels are are turned on and because the image is 2D the animation tapedeck is deactivated.

Here are a few extra notes on the functionality.

The standard viewerdisplaypanel control box control box is found along the left-hand side (zoom, pan, colormap control and region making).

There is an additional button at the bottom:

- Dismiss - dismiss the view. It can be recovered by executing the `view` function again (this is faster than if you destroyed (done) it).

The menus along the top are a standard service provided by the viewerdisplaypanel `tool`. See it for discussion of the menu items if they are unclear.

Note in particular, that the "DisplayData" menu item can be used to toggle on and off the raster and contour displays, regardless of the values you gave the view function for arguments `raster` and `contour`).

The animation tapedeck along the right is a standard service provided by the viewerdisplaypanel `tool`. See it for discussion of the tapedeck interface if is unclear.

- Vector maps displays

  If you set `vector=T` then the image will be displayed as a grid of vectors (all of the same amplitude). The image pixel values are interpreted as the position angle. If the image brightness unit (brightnessunit) is not angular, degrees will be assumed. You can set the brightness units with the Image `tool` function (setbrightnessunit)

  The position angle is measured positive North through East when you display a plane holding a celestial coordinate (the usual astronomical convention). For other axis/coordinate combinations, a positive position angle is measured from +x to +y in the absolute world coordinate frame.

  To make Vector map displays where the amplitude is also provided, you need a Complex image and the basic Viewer gui interface. For more details, see the discussion in the Getting Results document. See also the Imagepol `tool` which has a function makecomplex to make Complex images from amplitude/phase or real/imaginary Float images.

- The ImageAnalysis window

  The menu bar of the viewerdisplaypanel provides you with a submenu `Tools`. This has an item `ImageAnalysis`. If you select this menu item,

you are presented with some rollups which give access to image analysis functions, such as interactive regions, positions and profiles, statistics and summary. These rollups are described in the viewerdisplaypanel documentation.

- Connecting to the Regionmanager

  It is also possible to establish a connection with the Regionmanager `tool` so that it catches the interactive regions you create. This is done by requesting an 'interactive' region from the Regionmanager GUI. This action will start the `view` function of the specified (in the Regionmanager) image and the Regionmanager will collect the interactive regions that you make. The 'break' button will be enabled. If you wish to break the connection to the Regionmanager from the image display you can press the 'break' button on the `view` GUI.

- Events

  A number of events are generated by the `view` function and emitted by the Image `tool`. You can catch these yourself if you like, just like any *Glish* event.

Example

```
- im := image('hcn')
- whenever im->* do {
+  print $name, $value
+ }
```

This example catches all events from the Image `tool` **im** and prints out the event names and values.

Most of the events are simply forwarded from the viewerdisplaypanel, but image.view provides a few extra events.

- `region` see viewerdisplaypanel
- `position` see viewerdisplaypanel
- `statistics` see viewerdisplaypanel
- `breakfromviewer`

  This event is not of general user interest.

  The `activatebreak` argument of the `view` function is not intended for general user use (use the default value). However, if it is set to T, then the 'break' button on the displaypanel is activated. If it is set to F, the button is deactivated. If left `unset`, the button state is left unchanged. When the button is pressed, or when you change its

state via the call sequence, an event called `breakfromviewer` is emitted. The $value of the event are either T (break deactivated) or F (break activated).

– `viewerdone`

This event is not of general user interest.

The `view` function generates its own private Viewer `tool` with which it makes the display. When this private Viewer is destroyed (e.g. the user presses the done button) an event called `viewerdone` is emitted. Its $value is T.

Example

```
im := image('hcn')
im.view(raster=T, contour=T, adjust=T)
```

Displays the image as a raster and overlays contours. The adjustment panels (one for each display of the image) are shown as well.

1.2.2 is_image - Function

Package general
Module images

Is this variable an image tool ?

Synopsis
is_image(thing)

Arguments

thing	in	Variable to test Allowed:Glish variable Default: None

Returns
T or F

Description

Determine if this variable is an image tool. This is very useful in scripts; for robustness, you check that the tool is an Image `tool` before you start applying Image `tool functions`.

Example

```
- im:=image('myimage')
- is_image(im)
T
- im2 := [10,20]
- is_image(im2)
F
```

1.2.3 imagefiles - Function

Package general
Module images

Find the list of image files in the given directory

Synopsis
imagefiles(files, strippath, foreign)

Arguments

files	in	Path/files of interest Allowed:String Default: '.' the current working directory
strippath	in	Strip off leading path name ? Allowed:T or F Default: T
foreign	in	Include foreign formats with native support ? Allowed:T or F Default: F

Returns
Vector of Strings

Description

This (global) function returns a list of the names of the image files on disk in the specified path. You can provide just a directory, or a directory and wild-carded file list, or just the file list. You can optionally include the leading directory part of the output file names.
If `foreign==T`, then all foreign format image for which we have native access (FITS and Miriad presently) are included in the listing. Otherwise, and by default, just AIPS++ images are listed. Please note that presently, any FITS file (image or otherwise) will show up in the listing.

Example

```
- im1:=imagefromshape('zz', [10])
- im2:=imagefromshape('zz2', [10])
- im2.tofits('zz2.fits')
-
- imagefiles()
zz zz2
-
- imagefiles(foreign=T)
zz zz2 zz2.fits
- imagefiles(stripppath=F)
/u/nkilleen/zz /u/nkilleen/zz2
-
- imagefiles('./*2')                    # Wild card file name
zz2
```

1.2.4 imagetools - Function

Package general
Module images

Find the list of current global image tools

Synopsis
imagetools(showname, showclosed)

Arguments

showname	in	Show name of image assocoated with tool Allowed:T or F Default: F
showclosed	in	Include tool in list even if closed Allowed:T or F Default: T

Returns
Vector of Strings

Description

This (global) function retuns a list of the names of the image `tools` currently in Global name space. You can optionally list the names of the images (in parentheses) associated with the Image tools as well.

Example

```
- im1:=image('myimage')
- im2 := imagefromshape(shape=[10,10])
-
- imagetools()
im1 im2
- imagetools(showname=T)
im1(myimage) im2(Temporary_Image)
- im2.close()
- imagetools(showclosed=F)
im1
- imagedones()
```

1.2.5 imagedones - Function

Package general
Module images

Done all global or specified Image tools

Synopsis
imagedones(which)

Arguments

which	in	Which Image tools to destroy Allowed:String or vector of strings Default: All global Image tools

Returns
T or F

Description

This (global) function applies the done function to the specified Image `tools` currently in Global name space. By default it destroys all Image `tools`.

Example

```
- im1 := image('myimage1')
- im2 := image('myimage2')
- imagetools()
im1 im2
- imagedones();
Done applied to image tool "im1"
Done applied to image tool "im2"
#
- im1 := image('myimage1')
- im2 := image('myimage2')
- im3 := image('myimage3')
- imagedones("im1 im3")
Done applied to image tool "im1"
Done applied to image tool "im3"
```

1.2.6 imagedemo - Function

Package general
Module images

Demonstrate the use of an image

Synopsis

imagedemo()

Description

Demonstrate the use of the image class. This tool function is extremely rudimentary at the moment. It creates an image on disk called "demoimage.image" of shape [10,20,5] and filled with zeros. It then works through this image one plane at a time replacing the pixel values of that plane with the plane number. It then statistics evaluated over each plane. Finally, the image is converted to a FITS file called "DEMOIMAGE.FITS"

Returns
Bool - T means success

1.2.7 imagetest - Function

Package general
Module images

Run test suite for Image tool

Synopsis
imagetest(which)

Arguments

which	in	Which test to run Allowed:Integer or vector of integers Default: All tests

Returns
T or F

Description

Self-test of the Image `tool`. This function is quite extensive and includes forced errors. As long as the function finally returns with a `T`, it has succeeded (regardless of what error messages you might see).