1.2 images - Module

Access and analysis of images
include "images.g"

Description
This module contains functionality to access, create, and analyze AIPS++ images. It offers both basic services and higher-level packaged tools.
The available tools in this module are

- Image - create, manipulate and analyze images (no default tool). This `tool` provides a range of low to medium level services.
- Regionmanager - create and manipulate `regions-of-interest`(default tool is **drm**).
- Coordsys - create and manipulate Coordinate Systems (no default tool).
- Imagefitter - interactively fit 2-D models to an image (no default tool).
- Imageprofilefitter - interactively fit models to 1-D profiles from an image (no default tool).
- Imagepol - this offers specialized polarimetric analysis of images.
- Imageprofilesupport - this offers plotting of 1-D profiles with built in abcissa unit conversions. This is not a high-level user `tool`. It is a low-level `tool` used in high-level `tools` such as Imageprofilefitter.

Images
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. Some images don't have an associated disk file. These are called "virtual" images (e.g. an image that is temporary and entirely in memory). Whenever we use the word "image", we are just using it in a generic sense.
Images are manipulated with an Image tool.

Pixel mask
A `pixel mask` specifies which pixels are to be considered good (mask value `T`) or bad (mask 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.
An `image file` may contain zero, one, or more `pixel masks`. However, only one mask will be designated as the default mask and be applied to the data.

For more details, see the Image tool.

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` 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.

Coordinates
We will often refer to (absolute) pixel coordinates. Consider a 2-D image with shape [10,20]. Then our model is that the centre of the bottom-left corner pixel has pixel coordinate [X,Y] = [1,1]. The centre of the top-right corner pixel has pixel coordinate [X,Y] = [10,20].
When a physical Coordinate System (e.g. an RA/DEC direction coordinate) is attached to an image, then we can convert pixel coordinates to a world (or physical) coordinate.
The Coordsys `tool` is available for manipulating Coordinate Systems. You can recover the Coordinate System from an image into a Coordsys `tool` via the Image function coordsys.
For more details, see the Image and Coordsys tools.

Lattice Expression Language (LEL)
This 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 used in several of the Image tool functions.

Example

Here is a simple example to give you the flavour of using the `image` module. Suppose we have an image FITS disk file; we would like to convert it to an AIPS++ `image file` (stores the image in an AIPS++ table), look at the header information and store it in a *Glish* record for future use, work out some statistics and display the image with the Viewer.

```
include 'images.g'                                               # 1
im:=imagefromfits(outfile='data/ne.xyv', infile='$FITS/ne.FITS')    # 2
im.summary()                                                     # 3
im.statistics()                                                  # 4
```

```
im.view()                                                   # 5
im.done()                                                   # 6
```

1. This gives us access to the Image tool code.

2. Convert (using the `imagefromfits` constructor) an image FITS disk file called *'ne.FITS'* in the directory specified by the environment variable *'$FITS'* to an AIPS++ `image file` called *'ne.xyv'* in the subdirectory called *data*. The AIPS++ `image file` is now associated with an `image tool` called **im** which was constructed in this process.

3. This summarises (to the Logger) the basic header information in the image (name, masks, regions, brightness units, coordinates etc).

4. This `tool function` evaluates basic statistics from the entire image.

5. This displays the image with the Viewer.

6. This destroys the Image `tool`, but not its associated disk `image file`.

Tools

coordsys	Operations on CoordinateSystems
image	Operations on images
imagefitter	Fit models to images of the sky
imagepol	Polarimetric analysis of images
imageprofilefitter	Fit models to 1D profiles from images
imageprofilesupport	Plot profiles with axis label conversions
regionmanager	Create and manipulate regions of interest

Functions

coordsystest	Run test suite for Coordsys tool
coordsystools	Find the list of current global Coordsys tools
imagedemo	Demonstrate the use of an image
imagedones	Done all global or specified Image tools
imagefiles	Find the list of image files in the given directory
imagefittertest	Run test suite for Imagefitter tool
imagepoltest	Run test suite for Imagepol tool
imageprofilesupporttest	Run test suite
imagetest	Run test suite for Image tool
imagetools	Find the list of current global image tools
is_coordsys	Is this variable a Coordsys tool ?
is_image	Is this variable an image tool ?
is_imagepol	Is this variable an imagepol tool ?
is_region	Is this a valid region tool?

%

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 image to the rotated Coordinate System
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 image
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 file unaffected
- 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, tempclose=F)
```

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] [value=0.001, unit=Jy/beam]
```

---

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' in image 'myimage'
```

---

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

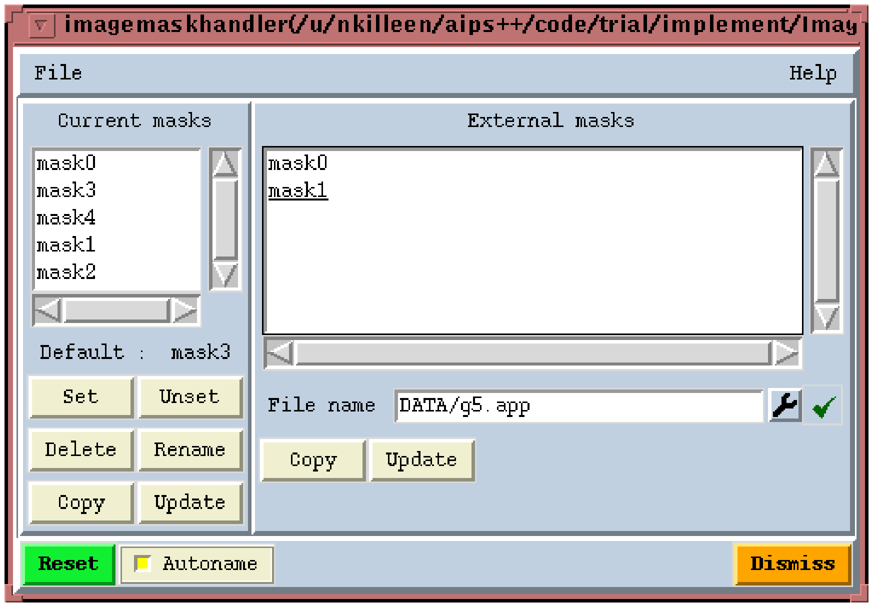

Figure 1.1: The maskhandler GUI

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=[10,20])
```

---

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 region
- im3 := im.subimage(outfile='/tmp/foo', region=r1)     # Explicitly named subimage
```

---

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 Units
------------------------------------------------------------------------------------------
1    Direction Right Ascension   SIN  512   10  00:00:00.000     6.00 -6.000000e+01 arcsec
2    Direction Declination       SIN  512   20 +00.00.00.000    11.00  6.000000e+01 arcsec
3    Spectral  Frequency             1024   30  1.415000e+09    16.00  1.000000e+03 Hz
               Velocity                         1.140944e+03    16.00 -2.110611e-01 km/s

- 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).

The function invokes many tests. The first one is a general test of everything at a basic level. The succeeding tests work on individual areas of the image module.
To run a specific test, you can use the `which` argument. Since the number of tests keeps growing, we don't tell you how many there are here or what exactly they do ! If you give too large a value, nothing will happen. `which` can be an integer or a vector of integers.

Example

```
- imagetest()
```

Runs all tests.

Example

```
- imagetest(10)
```

Runs test 10 only.

1.2.8 regionmanager - Tool

Package	**general**
Module	**images**

Create and manipulate regions of interest
include "regionmanager.g"

Constructors

regionmanager	Construct a regionmanager

Functions

absreltype	Convert region type value to a string
box	Create a pixel box region
complement	Create the complement of a world region
concatenation	Concatenate world regions along a new axis
copyregions	Copy regions from one Table to another
def	Default pixel coordinate value
deletefromtable	Delete regions from a Table
difference	Create the difference of two world regions

displayedplane	Find a region describing the plane displayed by the viewer
done	Destroy this regionmanager
extension	Extend a world region to extra dimensions
extractsimpleregions	Extract all simple regions into a record
fromglobaltotable	Save regions into a Table
fromrecordtotable	Save regions stored in a record into a Table
fromtabletoglobal	Restore a region from a Table to Glish
fromtabletonakedrecord	Restore all regions from a Table to a naked record
fromtabletorecord	Restore regions from a Table to a record
getselectcallback	Get the select callback function
gui	Start the GUI interface
intersection	Create the intersection of some world regions
ispixelregion	Is this region a pixel region ?
isworldregion	Is this region a world region ?
namesintable	Find the names of the regions stored in a Table
pixeltoworldregion	Convert a simple pixel region to a world region
pseudotoworldregion	Convert a Viewer pseudoregion to a world region
quarter	Create a quarter pixel region
setcoordinates	Set new default Coordinate System
setselectcallback	Set the callback function when GUI region selection is made
type	Return the type of this tool
union	Create a union of world regions
wbox	Create a world box region
wmask	Create a world mask region
wpolygon	Create a world polygon region with quantities
wrange	Create a world region for a pixel range on one axis

Overview of Regionmanager functionality

- Create simple pixel-coordinate based regions with functions

  box

  quarter

- Create simple world-coordinate based regions with functions

  wrange (range along one axis)

  wbox

  wpolygon

  wdbox

  wdpolygon

  Also related is function setcoordinates

- Create compound regions (from world regions only) with functions

complement
concatenation
difference
intersection
union

- The GUI interface is started with function gui.
- I/O of regions in tables with functions

copyregions
deletefromtable
namesintable
fromtabletoglobal
fromtabletorecord
fromtabletonakedrecord
fromglobaltotable
fromrecordtotable

- Convert pixel regions to world regions with

pixeltoworldregion.

- There are some general utility functions, generally only of interest if you are writing scripts. These are

absreltype
def
done
extractsimpleregions
ispixelregion
isworldregion
type

- There are some functions relating to interactive creation of regions with the Viewer. These are generally only of interest if you are writing scripts, and are

displayedplane
psuedotoworldregion

- There are some functions relating to communications. These are generally only of interest if you are writing scripts and are setselectcallback

getselectcallback

Regions and the Regionmanager

When working with an image, one is generally interested in some part of that image on which astrophysical analysis is performed. This `region-of-interest` (or more generically and simply, the 'region') might be the whole image, or some subset of it.

Regions come in a few varieties. There are simple regular shapes (box, ellipsoid), simple irregular shapes (polygon), as well as compound regions made by combining other regions (simple or compound). For example unions or intersections of regions. In addition, the simple regions can be pixel-coordinate based or world-coordinate based. However, a compound regions must always comprise world-coordinate based regions.

It is the task of the Regionmanager (`tool`) to manage your regions; it creates, lists and manipulates them. Apart from a Regionmanager, the only other way to create a `region-of-interest` is with the view function of the Image `tool`. This allows you to interactively make, with the cursor and an image display, a box or polygonal region. The region so made may be collected by the Regionmanager (in fact the complete process can be initiated from the Regionmanager).

Like many other AIPS++ tools, a default Regionmanager is created for you when you load its definition script (in this case, *regionmanager.g*). This pre-made tool is called **drm** (the defaultregionmanager). If it doesn't exist for some reason (you might have boldly deleted it maybe), then you can make another one with the regionmanager constructor. Although a Regionmanager `tool` can contain a little bit of state, in general, you should not need to make another one. The **drm** Regionmanager should be all you will ever need.

The Regionmanager has a custom GUI and a command line interface. It is probably fair to say that for the simplest regions, such as a pixel box, there is little to choose between making a region with the GUI or the command line interface. However, for the world regions, the GUI is significantly better; it hides the details of handling the coordinate information. Of course, the GUI always invokes the Regionmanager command line for you when it actually creates a region, but it removes the need for you to learn the Regionmanager command-line syntax.

Example

```
- include 'regionmanager.g'
- drm.type()
regionmanager
- drm.gui()
```

This loads the Regionmanager code and creates the default Regionmanager `tool` called **drm** - we ask it to tell us its type. We then start the GUI interface from which you can create all sorts of regions.

Simple Pixel and World regions
Pixel regions are specified purely in terms of pixel coordinates. Pixel coordinates are considered to run from 1 at the bottom left corner (blc) of an image to the image shape at the top-right corner (trc).
For example, a pixel region might be box that runs from a bottom blc of [20,20] to a trc of [64,90] in a 2D image. Such a pixel region isn't very portable though. If you are interested in a region about an astronomical source, pixel coordinates are only useful for a particular image. In some other image, the source may well have different pixel coordinates. However, pixel regions are easy to make and use, and they are valuable.
So far, we have been talking about absolute pixel coordinates. However, pixel regions can also be specified in relative coordinates. This is controlled through the argument `absrel` which may take on one of the values 'abs' (absolute coordinates) 'relref' (relative to the reference pixel of the image) or 'relcen' (relative to the center of the image). You can use the summary function of the image module to see the reference pixel. The sense of the offset is $rel = abs - ref$.
You may also define pixel regions in terms of fractional coordinates that are in the range [0,1] (blc to trc) - look for argument `frac` which can be T or F.

Example

From the command line, let's make a simple pixel box region with

```
- include 'image.g'
- im := image('myimage')
- const r1 := drm.box([10,20], [30,40])
- im.statistics(region=r1);
Number points = 441
Flux density  = -1.728895e-03  Jy
Sum           = -9.522969e-02       Mean     = -2.159403e-04
Variance      = 1.518106e-05        Sigma    = 3.896288e-03
Rms           = 3.897854e-03

Minimum value is -1.208747e-02  at [13, 32]
Maximum value is 1.287862e-02   at [21, 37]
```

You have now created a region tool called **r1** which describes a 2D box from [10,20] to [30,40] in absolute pixel coordinates. The region is made 'const' as well so you can't overwrite it by mistake. We then apply it to an image and evaluate some statistics in the specified region.

World regions are more portable, because the region is specified in terms of the world coordinates (e.g. RA and DEC, or Frequency etc). You can create a region with one image, and then apply it to another. For example, you might make an RA/DEC world box (i.e. blc and trc specified as RA and DEC) from an optical image. You can then apply it to your radio image of the same source. The software will look for the axes of the region (in this case RA and DEC), and then attempt to match them with the image to which the region is being applied. For each matching axis, the region will be applied.
When you make a world region, you must supply a coordinate system from the appropriate image to the Regionmanager. The coordinate system comes from the coordsys function of the Image tool. This associates a coordinate with an image pixel axis.

Example

Now let's do an example with a world box. It is better to make a world box with the GUI interface, but the command line is still manageable. First, let's summarise the header of our image (the one from the previous example).

```
- im.summary()
Image name       : myimage
Image mask       : Absent
Image units      : JY/BEAM

Direction system : J2000

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
T
```

And now we will make a world box with the blc at the reference pixel, and the trc offset somewhat. We use the quanta module for creating associations of values and units; in particular, the quantity function.

```
- blc := "17:42:29.303 -28.59.18.600";
- trc := "17:42:28.303 -28.59.10.600";
- cs := im.coordsys();
- r2 := drm.wbox(blc=blc,trc=trc,pixelaxes=[1,2],csys=cs)
- im.boundingbox(r2);
[blc=[90 90], trc=[103 98], inc=[1 1], bbShape=[14 9], regionShape=[14 9], imageShape=[155 178] ]
```

As well as passing in the blc and trc quantities, we have also told the Regionmanager which axes these values pertain to (they are numbered in the order in which they list in the `summary` function), and we also supplied it with

the coordinate system of the image. Note that with the GUI interface, both the axes and coordinates handling is hidden. If we wished, we could have left out the `pixelaxes` argument, and it would have defaulted to [1,2]. We then find the bounding box of the region when it's applied to the image. You can see from the summary listing that it is correct.
We generally think about our images as being in some order; RA/DEC/Frequency (for the X/Y/Z axes) or whatever. In the application of world regions, this order is unimportant. If you created a world region from an RA/DEC/Frequency image, and applied it to a Frequency/DEC/RA image, that image order change would be accounted for.
Note that for pixel regions however, because the region is not tagged with a coordinate system, the order of the image is relevant. So if you made a pixel box from an RA/DEC image and applied it to a DEC/RA image, the RA range would be applied to the DEC axis and vice versa. The pixel regions are present for simple usage only.

Simple pixel regions as world regions
It is possible to create a world region that is specified in pixel coordinates, and maintains the coordinate axis information. This can be useful because compound regions must be world-coordinate based (see below). We can do this because we have defined in the Regionmanager, "new" world units, called "pix" (which are then known to the quanta module) and these are recognized in the world regions. Similarly, we have also defined a unit called "frac". This allows you to specify world coordinates in units of the fraction of the image shape. For example, the blc of a 2D image has a "frac" coordinate of [0,0] and a trc of [1,1]. But be careful, such regions are still not portable to another image, because they are still effectively pixel-coordinate based (although masquerading as world regions).

Example

```
- include 'image.g'
- im := imagemaketestimage()
- cs := im.coordsys()
- local statsout
-
- r1 := drm.wbox(blc="10pix 10pix", trc="20pix 20pix", pixelaxes=[1,2], csys=cs)
- im.stats(statsout, region=r1, list=F, verbose=F)
- print 'Region 1: number points = ', statsout.npts
Region 1: number points =  121
-
- r2 := drm.wbox(blc="30pix 30pix", trc="40pix 40pix", pixelaxes=[1,2], csys=cs)
- im.stats(statsout,region=r2, list=F, verbose=F)
- print 'Region 2: number points = ', statsout.npts
Region 2: number points =  121
```

```
- r3 := drm.union(r1, r2)
- im.stats(statsout,region=r3, list=F, verbose=F)
- print 'Region 3: number points = ', statsout.npts
Region 3: number points =  242
```

In this example we make a union from two pixel boxes masquerading as world regions.

Compound Regions
It is often desirable to make a region which combines others. These are called compound regions. For example, give me the union of 3 regions and evaluate the statistics in that union. Or intersect this region with that one and extract the image data from that region. Compound regions are fully recursive; you can nest them as deeply as you like.
You *must* be aware that compound regions can only be made from world-coordinated based regions. You can convert a pixel region to a world region with the pixeltoworldregion function (at some point this function will move from the Regionmanager `tool` to the Image `tool`.

Example

```
- include 'image.g'
- im := image('myimage')
- cs := im.coordsys();
-
- blc := "17:42:29.303 -28.59.18.600";
- trc := "17:42:28.303 -28.59.10.600";
- r1 := drm.wbox(blc=blc,trc=trc,pixelaxes=[1,2],csys=cs);
- im.boundingbox(r1);
[blc=[90 90] , trc=[103 98] , inc=[1 1], bbShape=[14 9] , regionShape=[14 9] , imageShape=[155 178] ]
-
- blc := "17:42:27.303 -28.59.18.600"
- trc := "17:42:26.303 -28.59.10.600"
- r2 := drm.wbox(blc=blc,trc=trc,pixelaxes=[1,2],csys=cs);
- im.boundingbox(r2);
[blc=[116 90] , trc=[129 98] , inc=[1 1], bbShape=[14 9] , regionShape=[14 9] , imageShape=[155 178] ]
-
- r3 := drm.union(r1, r2)
- bb := im.boundingbox(r3)
- bb
[blc=[90 90] , trc=[129 98] , regionShape=[40 9] , imageShape=[155 178] ]
-
- im.statistics(statsout=s1, region=r1, list=F)
- im.statistics(statsout=s2, region=r2, list=F)
```

```
- im.statistics(statsout=s3, region=r3, list=F)
-
- np := prod((bb.trc-bb.blc) + 1)
- print 'No. pts in r1, r2, r3, bb(r3) = ', s1.npts, s2.npts, s3.npts, np
No. pts in r1, r2, r3, bb(r3) =  126, 126, 252, 360
-
-
- r4 := drm.intersection(r1, r2)
- im.boundingbox(r4)
SEVERE: Caught an exception! Event type=run exception=LCIntersection::LCIntersection - regions do not overlap
<fail>: Caught an exception! Event type=run exception=LCIntersection::LCIntersection - regions do not overlap
        File:   servers.g, Line 711
        Stack:  .()
                .()
```

We make two world boxes and find the bounding box of each. Then we make the union of the two and find the bounding box which we store in a record called `bb`. We then find some statistics over the union (supressing the logger output and storing the results in records). You can see that the number of points found in the region reflects the union, not the bounding box of the union.
Finally, we make an intersection of our two regions. Because regions r1 and r2 don't actually intersect, a fail is generated.

Automatic Extension
One general philosophy behind the regions software is that if you apply a region to an image, but don't explicitly specify all axes of the image in that region, then for the unspecified axes, the full image is taken. For example, imagine that from an RA/DEC optical image you made an RA/DEC region. You then applied it to an RA/DEC/Frequency radio spectral-line cube. For the frequency axis, which was not specified in the region, all frequency pixels would be selected. Another philosophy is that if, on application, a region extends beyond an image, the overhang is silently discarded.

Example

```
- include 'image.g'
- im.boundingbox(drm.box())
[blc=[1 1] , trc=[155 178] , inc=[1 1] , bbShape=[155 178] , regionShape=[155 178] , imageShape=[155 178] ]
-
- r1 := drm.box(trc=[20])
- im.boundingbox(r1)
[blc=[1 1] , trc=[20 178] , inc=[1 1] , bbShape=[20 178] , regionShape=[20 178] , imageShape=[155 178] ]
-
```

```
- r2 := drm.box(trc=[9000,20])
- im.boundingbox(r2)
[blc=[1 1] , trc=[155 20] , inc=[1 1] , bbShape=[155 20] , regionShape=[155 20] , imageShape=[155 178] ]
```

In the first example, none of the axes are specified in the region, so it defaults to the full image on all axes. In the second example only the first axis of the trc is specified. In the third example, the trc overhang is silently discarded.

Masks
Some comment about the combination of masks and `regions-of-interest` is useful here. 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 mask of an image from a polygonal `region-of-interest`. Now, necessarily, the mask is returned to you in regular Boolean array. Thus, the array shape 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).

Vector Inputs
Many of the functions of a Regionmanager `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').

Default Values
When specifying blc and trc vectors for some regions (usually boxes) it can often be that you wish to default one axis but not others.
For example, you may like to specify a value for axis 2 but leave axis 1 at its defaults. This is trivial with the GUI interface (you just don't fill in the ones you are not interested in), but is a little harder with the command line interface.
There is a function called def which provides a magic value which can be used with pixel boxes for this purpose. For example,

```
- include 'image.g'
- im := imagefromshape('x', [20,30])
- r := drm.box(trc=[drm.def(),10])
- im.boundingbox(region=r)
[blc=[1 1] , trc=[20 10] , regionShape=[20 10] , imageShape=[20 30] ]
```

You can see that trc for axis 1 has defaulted to the shape of the image.
With world boxes, we have defined a magic unit called 'default' instead. Thus

```
- include 'image.g'
- im := imagefromshape('x', [20,30])
- cs := im.coordsys()
- drm.setcoordinates(cs)
- r := drm.wbox(trc="0default 10pix")
- im.boundingbox(region=r)
[blc=[1 1] , trc=[20 10] , regionShape=[20 10] , imageShape=[20 30] ]
```

Error Checking
Although some error checking is done when the region is created (in *Glish*), much of it does not occur until the region is applied to an image. In particular, when creating a compound region, it is *not* checked that the compound region contains only world-coordinate based regions. It is only when you apply the region to an image that these checks are made. Any errors that occur will cause exceptions to be generated. Thus, when writing scripts, you should always check for a fail status for when you create or use a region.

Example

```
- include 'image.g'
- const stats := function(imageobject, blc, trc, csys)
{
  r1 := drm.wbox(blc=blc, trc=trc, csys=csys)
  if (is_fail(r1)) {
     fail
  } else {
     ok := imageobject.statistics(r1)
     if (is_fail(ok)) fail
  }
  return T
}
```

This fairly silly function detects if the region creation failed (function given invalid arguments) and then returns the fail condition. Statistics are then evaluated from the region and the fail status checked in case the region was invalid (or for some other reason).

How a region is stored in AIPS++

In AIPS++, a region is itself a `tool`. This means it can be transmitted about with *Glish*, saved to Tables and restored from Tables. A region is actually made with a generic container `tool` called an Itemcontainer. The regions that you make have no intelligence regarding their greater purpose in life (although of course an Itemcontainer does have some functions so they can be manipulated). All of the knowledge about regions lies with the Regionmanager. It should not be necessary for you to know anything about the insides of regions. You just need to know how to make them and how to use them.

Example

```
- include 'regionmanager.g'
- r1 := drm.box([10,20], [30,40])
- r1
[gui=<function>, done=<function>, get=<function>, has_item=<function>,
makeconst=<function>, length=<function>, names=<function>,
set=<function>, add=<function>, fromrecord=<function>,
torecord=<function>, type=<function>]
-
- r1.torecord()
[name=LCSlicer, isRegion=3, blc=[10 20] , trc=[30 40] , inc=,
fracblc=[F F] , fractrc=[F F] , fracinc=,
arblc=[1 1] , artrc=[1 1] , oneRel=T, comment=]
```

First we make a simple pixel-box region. We then try and print it out. You can see that this isn't very enlightening. What you see are the Itemcontainer's functions. To really see inside the region, you can use the torecord function of Itemcontainer. This converts the region to a record which you can view if you really must (the record is of no functional use when dealing with regions). It's perhaps also useful to know that once you have made a region you can't break it. You can delete it, maybe overwrite it, but you can't mistakenly mess up its contents (even with the Itemcontainer functions with which it was created). It's user proof - there's a challenge for you !

regionmanager - Constructor

Package	**general**
Module	**images**
Tool	**regionmanager**

Construct a regionmanager

Synopsis
regionmanager()

Returns
Regionmanager tool

Description

This is the only Regionmanager constructor. It should generally be unnecessary for you to make one as there is little state in a Regionmanager (you can set a Coordinate System with setcoordinates); the default Regionmanager **drm** should be all you need.

Example

```
- include 'regionmanager.g'
- const myrm := regionmanager()
- myrm.type()
regionmanager
```

---

regionmanager.absreltype - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Convert region type value to a string

Synopsis
absreltype(absrelvalue)

Arguments

absrelvalue	in	Region type value Allowed:Integer - 1, 2, 3, 4

Returns
String or fail

Description

This function is not intended for general user use.
Regions may be specified with coordinates which are absolute or relative. This function converts the integer code defining the absolute/relative type of the coordinates (which is stored in the region) into a string (maybe for printing purposes).
The different types are

```
Integer     String      Description
1            abs        Absolute coordinate
2            relref     Relative reference pixel
3            relcen     Relative to center of image
4            reldir     Relative to some direction
```

Example

```
- r := drm.box(blc=[3,40], trc=[80,90])   # Create region
- v := r.get('arblc')                     # Get absrel value vector for blc
- n := length(v)
- for (i in 1:n) {
+   print drm.absreltype(v[i])            # Print string conversion for each axis
+  }
-
```

---

regionmanager.box - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create a pixel box region

Synopsis
box(blc, trc, inc, absrel, frac, comment)

Arguments

blc	in	blc of the box Allowed:Numeric vector Default: Unity
trc	in	trc of the box Allowed:Numeric vector Default: Shape
inc	in	increment Allowed:Numeric vector Default: unity
absrel	in	Absolute or relative coordinates Allowed:One of 'abs', 'relref' or 'relcen' Default: 'abs'
frac	in	Pixel or fractional coordinates Allowed:T or F Default: F
comment	in	A comment stored with the region Allowed:String

Returns
Region tool or fail

Description

This function creates a multi-dimensional pixel box region. The box is specified by a bottom-left corner, and top-right corner and an increment (or stride). Pixel coordinates are considered to run from 1 at the bottom left corner of the image to the image shape at the top-right corner of the image. You can specify whether the coordinates are given as pixel coordinates (`frac=F`) or fractions of the image shape (`frac=T`). Absolute fractions are in the range [0,1].
You can also specify whether the coordinates are given as absolute coordinates (`absrel='abs'`) or relative to the reference pixel (`absrel='relref'`) or relative to the center of the image (`absrel='relcen'`).

Example

```
- im := image('myimage')
- im.shape()
[155 178 256]
-
- r := drm.box()                 # create region
-
```

```
- im.bb(r)
[blc=[1 1 1] , trc=[155 178 256] , inc=[1 1 1] , bbShape=[155 178 256] ,
regionShape=[155 178 256] , imageShape=[155 178 256] ]
```

This region, on application to an image, selects the entire image.

Example

```
- im := image('myimage')
- im.shape()
[155 178 256]
-
- r := drm.box(blc=[5,10])
- im.bb(r)
[blc=[5 10 1] , trc=[155 178 256] , inc=[1 1 1] , bbShape=[151 169 256] ,
regionShape=[151 169 256] , imageShape=[155 178 256] ]
```

This region is only specified for the first two axes of the blc. Automatic extension rules apply for the other axis and the trc (defaults to the shape).

Example

```
- im := image('myimage')
- im.shape()
[155 178 256]
-
- r := drm.box(blc="10 10 10", trc="20 20 20", inc="2 2 2")
- im.boundingbox(r)
[blc=[10 10 10] , trc=[20 20 20] , inc=[2 2 2] , bbShape=[11 11 11] ,
regionShape=[6 6 6] , imageShape=[155 178 256] ]
-
- im.stats(stats, region=r, list=F);
- stats.npts
216
```

This region picks out every other pixel in the 3D box. The “regionShape” field of the bounding box record does reflect the increment whereas “bbShape” does not. You can see that the number of points used in determining the statistics (216) reflects the increment as well.

Example

```
- im := image('myimage')
- im.shape()
[64 128]
-
- rmd := drm.def()
- r := drm.box([-5,-10], [rmd,20], absrel='relcen')
-
- im.boundingbox(r)
[blc=[28 55] , trc=[64 85] , inc=[1 1] , bbShape=[37 31] ,
regionShape=[37 31] , imageShape=[64 128] ]
```

The region is specified in pixels relative to the center of the image. Note the use of the default value (`drm.def()`) to default the first axis of the trc argument to the image shape without having to know the image shape.

Example

```
- im := image('myimage')
- im.shape()
[155 178 256]
-
- im.summary(summ, list=F)
- summ.refpix
[90 90 1]
-
- r := drm.box([-0.25,-0.3], [0.25, 0.5], frac=T, absrel='relref')
- im.bb(r)
[blc=[39 37 1] , trc=[116 178 256] , inc=[1 1 1] , bbShape=[78 142 256] ,
regionShape=[78 142 256] , imageShape=[155 178 256] ]
```

This example shows selection by relative to reference pixel fractional coordinates plus auto extension to unspecified axes.

---

regionmanager.complement - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create the complement of a world region

Synopsis
complement(region, comment)

Arguments

region	in	The world region Allowed:Region tool
comment	in	A comment stored with the region Allowed:String

Returns

Region tool or fail

Description

This function (short-hand name `comp`) creates the complement of a world region.

Example

```
- im := image('hcn')
- cs := im.coordsys()
- im.shape()
[155 178]
-
- blc := "17:42:29.303 -28.59.18.600"
- trc := "17:42:28.303 -28.59.10.600"
- r2 := drm.wbox(blc,trc,[1,2],cs)
- r3 := drm.complement(r2);
-
- im.stats(region=r2)                          # Some output discarded
Selected bounding box [90, 90] to [103, 98]
No pts   = 126
-
- im.stats(region=r3)
Selected bounding box [1, 1] to [155, 178]     # Some output discarded
No pts   = 27464
```

As expected, the number of pixels in the complement is $(155 * 178) - 126 = 27464$

---

regionmanager.concatenation - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Concatenate world regions along a new axis

Synopsis
concatenation(box, regions, comment)

Arguments

box	in	The axis to concatenate along Allowed:world box region
regions	in	World regions Allowed:Record containing world regions
comment	in	A comment stored with the region Allowed:String

Returns
Region tool or fail

Description

This function (short-hand name `concat`) creates a region which is the concatenation along a new axis of the given world regions.
This function is similar to the extension function. The `concatenation` function allows you to take many world regions, and concatenate them along one axis (rather than take one region and extend it along many axes which is what function `extension` does).
For example, you may have generated a different polygonal region for each spectral pixel of a spectral-line cube and you wish to concatenate them together to form the overall region for use in a deconvolution application.
The axis to concatenate along is specified as a 1-dimensional world box. The shape of the 1D box must contain as many pixels (although you don't have to specify it in pixels) as there are regions to concatenate.
Because this function is most likely to be used in a script, the interface takes a record containing `region` tools, rather than the names of the tools themselves (there might be a lot of them).

Example

```
- im := image('cube')
- cs := im.coordsys()
- drm.setcoordinates(cs, verbose=F)      # Don't tell us each time
```

```
                                         # private coordinates used
- box := drm.wbox(blc="20pix", trc="25pix", pixelaxes=[3])
- bb := im.boundingbox(box)
-
- rec := [=];
- local x, y;
- j := 1;
- for (i in bb.blc[3]:bb.trc[3]) {
+ # Some code in function ‘mypolygon’ generates the
+ # x and y vectors for this spectral pixel, perhaps interactively
+
+    mypolygon(x,y);
+    rec[j] := drm.wpolygon(x,y,[1,2])
+    j +:= 1;
+  }
- rc := drm.concat(box, rec)
-
- im.stats(region=rc, axes=[1,2])
Plane Freq         Npts Sum            Mean           Rms            Sigma          Minimum        Maximum
20    1.413724e+09 25   -4.778154e+00  -1.911262e-01  2.578399e-01   1.766359e-01   -4.252437e-01  1.820721e-01
21    1.413744e+09 40   -7.476902e+00  -2.990761e-01  3.692736e-01   2.210687e-01   -6.073643e-01  1.634156e-01
22    1.413763e+09 32   -2.696485e+00  -1.078594e-01  1.916686e-01   1.617070e-01   -3.295788e-01  1.422531e-01
23    1.413783e+09 77   4.889158e-01   1.955663e-02   3.148451e-02   2.518293e-02   -3.671944e-02  6.521463e-02
24    1.413803e+09 25   -1.337832e+00  -5.351327e-02  6.296221e-02   3.385893e-02   -1.232493e-01  1.014871e-02
25    1.413823e+09 15   1.091297e+00   4.365189e-02   7.252339e-02   5.910932e-02   -6.364560e-02  1.630955e-01
```

In this example, we create a 1D box and use it to concatenate 2D xy polygons along the z axis. We then ask for the statistics of each plane in the region. There is a differerent number of pixels per plane as each polygon is different.

---

regionmanager.copyregions - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Copy regions from one Table to another

Synopsis

copyregions(tableout, tablein, confirm)

Arguments

tableout	inout	The output table Allowed:Image tool, table tool or String
tablein	in	The input table Allowed:Image tool, table tool or String
confirm	in	Prompt for confirmation Allowed:T or F Default: F

Returns

Bool or fail

Description

This function copies all the regions from one AIPS++ Table to another.
For the Table `tool` arguments, you can specify an image tool, a table tool, or a string. If you give a string, it should be the name of an existing AIPS++ table on disk (any kind of table) or a new table (for `tableout`) which will be created for you. The new table has no useful structure beyond housing the regions for you.
If desired, you can request confirmation prompting to prevent accidental overwrite of existing regions.

Example

```
- im := image('hcn')
- drm.copyregions('newtable', im)
```

Creates the output table and copies all the regions housed by the image *hcn* via the Image `tool` **im**.

---

regionmanager.def - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Default pixel coordinate value

Synopsis

def()

Returns
Integer

Description

This function returns a value which can be used as a default when creating pixel regions. For example, you might wish to make a pixel box and default the value of the first axis of the `trc` argument (to the shape) but specify the value for the second axis.

Example

```
- im := image('quintin')
- im.shape()
[155 178 256]
-
- blc := [12,drm.def(),43]
- trc := [drm.def(),50]
- r1 := drm.box(blc, trc)
-
- im.boundingbox(r1)
[blc=[12 1 43] , trc=[155 50 256] , inc=[1 1 1] , bbShape=[144 50 214] ,
regionShape=[144 50 214] , imageShape=[155 178 256] ]
```

So you can see that the `blc` and `trc` elements which were given the value `drm.def()` have been given default values; 1 for the `blc` and `shape=128` for the `trc`. Note that `trc[3]` was not given. It has been returned as 256. It didn't need a `drm.def()` because it was the last dimension.

Example

Typing `drm.def()` is rather cumbersome. However, it is unique. To make it easier, you could define your own symbol according to

```
- const rmd := drm.def()
- blc := [1,23,43]
- trc := [rmd,50]
- r1 := drm.box(blc, trc)
```

---

regionmanager.deletefromtable - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Delete regions from a Table

Synopsis
deletefromtable(tablename, confirm, regionname)

Arguments

tablename	in	The table Allowed:Image tool, table tool or String
confirm	in	Prompt for confirmation Allowed:T or F Default: F
regionname	in	Name(s) of the region(s) to delete Allowed:Vector of strings

Returns
Bool or fail

Description

This function deletes regions stored in an AIPS++ Table.
For the `tablename` argument, you can specify an image tool, a table tool, or a string. If you give a string, it should be the name of an existing AIPS++ table on disk (any kind of table).
You specify the name(s) of the regions with the `regionname` arguments. If you set `regionname='all'` then all regions are deleted. The names of all the regions stored in a Table can be found with the function namesintable.

Example

```
- im := image('hcn')
- names := drm.namesintable(im)
- drm.deletefromtable(im, F, names)
```

In this example, we delete all of the regions in a table, if there are any. The regions are deleted without confirmation prompting.

regionmanager.difference - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create the difference of two world regions

Synopsis
difference(region1, region2, comment)

Arguments

region1	in	The first world region Allowed:Region tool
region2	in	The second world region Allowed:Region tool
comment	in	A comment stored with the region Allowed:String

Returns
Region tool or fail

Description

This function (short-hand name `diff`) creates a region which is the difference of two world regions. The order of the regions is important.

Example

```
- im := image('hcn')
- cs := im.coordsys()
- drm.setcoordinates(cs)
-
- blc := "10pix 10pix"
- trc := "60pix 60pix"
- r1 := drm.wbox(blc,trc,[1,2])
-
- blc := "50pix 50pix"
- trc := "80pix 80pix"
```

```
- r2 := drm.wbox(blc, trc, [1,2])
-
- r3 := drm.difference(r1, r2)                # r1 - r2
-
- im.stats(region=r1)                         # Some output discarded
Selected bounding box [10, 10] to [60, 60]
No pts   = 2601
-
- im.stats(region=r3)                         # Some output discarded
Selected bounding box [10, 10] to [60, 60]
No pts   = 2480
-
-
- r4 := drm.difference(r2, r1)                # r2 - r1
-
- im.stats(region=r2)                         # Some output discarded
Selected bounding box [50, 50] to [80, 80]
No pts   = 961
-
- im.stats(region=r4)                         # Some output discarded
Selected bounding box [50, 50, 1] to [80, 80, 64]
No pts   = 840
```

We use pixel units and boxes in this example to make it clear what is happening. The two regions overlap in the top right corner area of region **r1** by an area of $11 \times 11 = 121$ pixels. Therefore, the difference region **r3** has $2601 - 121 = 2480$ pixels in it. For difference region **r4**, the region of overlap is the bottom left corner area of region **r2** and still contains 121 pixels. We expect $961 - 121 = 840$ pixels in the differnce region.

---

regionmanager.displayedplane - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Find a region describing the plane displayed by the viewer

Synopsis

displayedplane(image, ddoptions, zaxispixel, asworld)

Arguments

image	in	Image Allowed:Image tool
ddoptions	in	Options from display data Allowed:Record
zaxispixel	in	The pixel of the z (movie) axis Allowed:Integer Default: Unset
asworld	in	T for a world box, F for a pixel box Allowed:T or F Default: T

Returns
region tool or fail

Description

This function is for use with the viewer and is not of general user interest. This function will return a world region describing the plane of the image currently displayed by the viewer. Note that this region describes the full plane of the image, not just the piece you might have given to the viewer (see example). You must provide

- `image` This is the Image `tool` that the conversion pertains to.
- `ddoptions` is a record obtained from the Viewerdisplaydata `tool` via the getoptions function. It contains information about what is displayed.
- `zaxispixel` is the current pixel of the movie or Z axis. This is the axis that the viewer animator steps along. It can be obtained from the index function. If it's unset, then the region will not be intersected with the present movie axis plane. It will always be intersected with the hidden axes planes.

Example

```
- include 'image.g'
- include 'viewer.g'
- include 'regionmanager.g'
#
- im : =image('myimage')
- print im.shape()
[10 20 4 5]
- r1 := drm.quarter()
```

```
- im.bb(r1)
[blc=[4 6 1 1] , trc=[8 15 4 5] , inc=[1 1 1 1] , bbShape=[5 10 4 5] ,
regionShape=[5 10 4 5] , imageShape=[10 20 4 5] ]
#
- displaypanel := dv.newdisplaypanel()
- displaydata := dv.loaddata(im.name(F), 'raster')
- displaydata.setoptions([region=r1])
- displaypanel.register(displaydata)
#
- options := displaydata.getoptions()
- plane := drm.displayedplane(im, options, displaypanel.index())
- if (is_fail(plane)) fail
- im.bb(region=plane)
[blc=[1 1 1 1] , trc=[10 20 1 1] , inc=[1 1 1 1] , bbShape=[10 20 1 1] ,
regionShape=[10 20 1 1] , imageShape=[10 20 4 5] ]
-
- cs := im.coordsys()
- dr := drm.intersection(drm.pixeltoworldregion(csys=cs,region=r1), plane)
- im.bb(dr)
[blc=[4 6 1 1] , trc=[8 15 1 1] , inc=[1 1 1 1] , bbShape=[5 10 1 1] ,
regionShape=[5 10 1 1] , imageShape=[10 20 4 5] ]
```

In this example, we use the default Viewer (**dv**) to create a **Viewerdisplaypanel** on which we register a **Viewerdisplaydata** which will display the specified image as a raster. A region specifying a quarter of the image is given. The region describing the plane being displayed can be seen to reflect the full shape of the image. To make a region describing what is actually being displayed, one can simply intersect it with the region given to the viewer, as shown. Note the conversion of the pixel box to a world box before it can be used in the compound region.

---

regionmanager.done - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Destroy this regionmanager

Synopsis
done()

Returns

Bool or fail

Description

This function destroys the contents of the **regionmanager** `tool` (including its GUI). The `tool` still exists as a Glish variable, but it is no longer a Regionmanager ! You are unlikely to need this function.

Example

```
- const myrm := regionmanager()
- myrm.done()
T
- print 'myrm=', myrm
myrm=F
```

You can see that after calling the **done** function the `myrm` variable still exists, but its value is F.

---

regionmanager.extension - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Extend a world region to extra dimensions

Synopsis

extension(box, region, comment)

Arguments

box	in	The world region to extend over Allowed:World box region tool
region	in	The world region to extend Allowed:Region tool
comment	in	A comment stored with the region Allowed:String

Returns

Region tool or fail

Description

This function (short-hand name `ext`) creates a region which is the extension over specified dimensions along straight lines of the given region. For example, you might like to extend a 2-D polygon to be a 3-D 'cylindrical polygon' (that is, the polygon is replicated for each plane).
The extension is specified with a world box. It can be of any dimension. However, you can't put the same axes in the extension box as are in the region you are extending.

Example

```
- im := image('cube')
- cs := im.coordsys()
- im.shape()
[155 178 64]
-
- drm.setcoordinates(cs)
-
- x := dq.quantity([40,45,48,55,48,43,40], 'pix')
- y := dq.quantity([40,43,59,60,53,48,45], 'pix')
- poly := drm.wpolygon(x, y, [1,3])
- box := drm.wbox(dq.quantity("20pix"), dq.quantity("30pix"), [2])
-
- ext := drm.extension(box, poly, 'a cylindrical polygon')
-
- im.boundingbox(poly)
[blc=[40 1 40] , trc=[55 178 60] , inc=[1 1 1] , bbShape=[16 178 21] ,
regionShape=[16 178 21] , imageShape=[155 178 256] ]
-
- im.boundingbox(box)
[blc=[1 20 1] , trc=[155 30 256] , inc=[1 1 1] , bbShape=[155 11 256] ,
regionShape=[155 11 256] , imageShape=[155 178 256] ]
-
- im.boundingbox(ext)
[blc=[40 20 40] , trc=[55 30 60] , inc=[1 1 1] , bbShape=[16 11 21] ,
regionShape=[16 11 21] , imageShape=[155 178 256] ]
```

In this example we create a 2D polygon in the XZ plane. We then create a 1D box along the Y axis from pixels 20 to 30. Then we extend the polygon with that box to make a series of XZ polygons for the specified Y pixels.

Note that when the `box` and `poly` regions are applied to the image with the `boundingbox` function, the auto-extension operation is applied. It is not until the region has as many dimensions as the image (the region `ext`) that no auto-extension occurs.

---

regionmanager.extractsimpleregions - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Extract all simple regions into a record

Synopsis
extractsimpleregions(region)

Arguments

region	in	The region to extend Allowed:Region tool

Returns
Record of Region tools or fail

Description

This function (short-hand name `esr`) extracts all simple (i.e. non-compound) regions from the given region `tool`. This might be useful to plot all of the simple regions that made up a compound region.

Example

```
- im := image('cube')
- cs := im.coordsys()
- drm.setcoordinates(cs)
-
- x := dq.quantity([40,45,48,55,48,43,40], 'pix')
- y := dq.quantity([40,43,59,60,53,48,45], 'pix')
- poly := drm.wpolygon(x, y, [1,3])
```

```
- box := drm.wbox(dq.quantity("20pix"), dq.quantity("30pix"), [2])
-
- union := drm.union(box, poly);
- rec := drm.extractsimpleregions(union)
- print 'Number of extracted simple regions = ', length(rec)
2
```

---

regionmanager.fromglobaltotable - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Save regions into a Table

Synopsis
fromglobaltotable(tablename, confirm, verbose, regionname, ...)

Arguments

tablename	inout	The table Allowed:Image tool, table tool or String
confirm	in	Prompt for confirmation Allowed:T or F Default: F
verbose	in	Report successful saves Allowed:T or F Default: T
regionname	in	Name(s) of the region(s) when saved in the table Allowed:Vector of strings Default: Self naming
...	in	Region(s) to save Allowed:Region tool(s)

Returns
Bool or fail

Description

This function saves regions into an AIPS++ Table.

For the `tablename` argument, you can specify an image `tool`, a table `tool`, or a string. If you give a string, it should be the name of an existing AIPS++ table on disk (any kind of table) or a new table which will be created for you. The new table has no useful structure beyond housing the regions for you. You can specify the name that each region will have (`regionname`) when it is saved in the Table. If you don't specify this, a name will be made up for you (region1, region2, etc).

Example

```
- im := image('hcn')
- r1 := drm.box()
- r2 := drm.quarter()
- drm.fromglobaltotable(im, T, F, "x1 x2", r1, r2)
- drm.namesintable(im)
x1 x2
```

In this example, we save two regions called **r1** and **r2** to the table (previously containing no regions) referred to by the image `tool` **im**. The regions are renamed to 'x1' and 'x2' as they are stored.

Example

```
- im := image('hcn')
- r1 := drm.box()
- r2 := drm.quarter()
- drm.fromglobaltotable(im, T, F, "", r1, r2)
- drm.namesintable(im)
x1 x2
```

In this example, we save two regions called **r1** and **r2** to the table (previously containing no regions) referred to by the image tool **im**. The names for regions are made up for us as we don't specify them. Note that because the regions are specified by the special *Glish* '...' argument (it has no actual argument name), we must give the `regionname` argument explcitly as an empty vector of strings (else *Glish* will take the empty string as a region).

---

regionmanager.fromrecordtotable - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Save regions stored in a record into a Table

Synopsis
fromrecordtotable(tablename, confirm, verbose, regionname, regionrec)

Arguments

tablename	inout	The table Allowed:Image tool, table tool or String
confirm	in	Prompt for confirmation Allowed:T or F Default: F
verbose	in	Report successful saves Allowed:T or F Default: T
regionname	in	Name(s) of the region(s) when saved in the table Allowed:Vector of strings
regionrec	in	Region(s) to save Allowed:Record of region tool(s)

Returns
Bool or fail

Description

This function saves regions into an AIPS++ Table. It is just like tool fromglobaltotable except that the input regions are stored in a record rather than existing in global name space.
For the `tablename` argument, you can specify an image `tool`, a table `tool`, or a string. If you give a string, it should be the name of an existing AIPS++ Table on disk (any kind of table) or a new Table which will be created for you. The new table has no useful structure beyond housing the regions for you.
You can specify the name that each region will have (`regionname`) when it is saved in the Table. If you don't specify this, the field name of the corresponding region in the record is used.

Example

```
- im := image('hcn')
- r := [=]
```

```
- r[1] := drm.box()
- r[2] := drm.quarter()
- drm.fromrecordtotable(im, T, F, "x1 x2", r)
- drm.namesintable(im)
x1 x2
```

In this example, we save two regions stored in the record `r` to the table (previously containing no regions) referred to by the Image `tool` **im**. The regions are renamed to 'x1' and 'x2' as they are stored.

Example

```
- im1 := image('hcn')
- im2 := image('hcn2')
- r := drm.fromtabletorecord(im1)
- names := drm.namesintable(im1)
- drm.fromrecordtotable(im2, T, F, names, r)
```

In this example, we recover the regions into a record from one image, and then copy them to another.

---

regionmanager.fromtabletoglobal - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Restore a region from a Table to Glish

Synopsis

fromtabletoglobal(tablename, verbose, regionname)

Arguments

tablename	in	The table Allowed:Image tool, table tool or String
verbose	in	Report successful restores Allowed:T or F Default: T
regionname	in	Name of the region to restore Allowed:String

Returns
Region tool or fail

Description

This function restores a region from an AIPS++ Table to the global name space of *Glish*.
For the `tablename` argument, you can specify an image `tool`, a table `tool`, or a string. If you give a string, it should be the name of an existing AIPS++ table on disk (any kind of table).
You can use the function namesintable to find out the names of the regions in the Table.

Example

```
- im := image('hcn')
- names := drm.namesintable(im)
  for (i in 1:length(names)) {
+    cmd := spaste(names[i], ' := drm.fromtabletoglobal(im, regionname=\'', names[i], '\')')
+    print 'cmd=', cmd
+    eval(cmd)
+ }
cmd= x1 := drm.fromtabletoglobal(im, regionname='x1')
Successful restore of region x1
cmd= x2 := drm.fromtabletoglobal(im, regionname='x2')
Successful restore of region x2
```

We find out the names of the regions in the Table, and then restore them in a loop with the *Glish* command `eval`; it creates the region with the name given in the array `names`. Because the output region name is not known to the function **fromtabletoglobal**, it cannot prompt for confirmation if the symbol already exists in *Glish*. That's your job.

---

regionmanager.fromtabletorecord - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Restore regions from a Table to a record

Synopsis
fromtabletorecord(tablename, verbose, regionname, numberfields)

Arguments

tablename	in	The table Allowed:Image tool, table tool or String
verbose	in	Report successful restores Allowed:T or F Default: T
regionname	in	Name of the region(s) to restore Allowed:Vector of strings Default: All
numberfields	in	Number the fields ? Allowed:T or F Default: T

Returns
Record or fail

Description

This function restores a region from an AIPS++ Table to the global name space.
For the `tablename` argument, you can specify an image `tool`, a table `tool`, or a string. If you give a string, it should be the name of an existing AIPS++ table on disk (any kind of table).
If `numberfields` is F, then the field names of the record are the same as they are in the Table. Otherwise, the regions are put into numbered fields (the field names could be anything).
You can use the function namesintable to find out the names of the regions in the Table.

Example

```
- im := image('hcn')
- rec := drm.fromtabletorecord(im, numberfields=T)
- print is_region(rec[1])
```

The record fields are numbered, not named.

---

regionmanager.fromtabletonakedrecord - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Restore all regions from a Table to a naked record

Synopsis

fromtabletonakedrecord(tablename)

Arguments

tablename	in	The table Allowed:Image tool, table tool or String

Returns

Record or fail

Description

This function restores all the regions from an AIPS++ Table to a naked record. By that we mean that the regions themselves are exposed by conversion to a record (which you can always do with function torecord) and then all the region records are placed into a record themselves. The field names of the record are the names of the regions as stored in the Table.
For the `tablename` argument, you can specify an image `tool`, a table `tool`, or a string. If you give a string, it should be the name of an existing AIPS++ table on disk (any kind of table).
This function is not intended for general user use since you should not generally need to break the encapsulation of a region.

Example

```
- im := image('hcn')
- rec := drm.fromtabletonakedrecord(im)
- field_names(rec)
r1 r2 r3
```

---

regionmanager.getselectcallback - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Get the select callback function

Synopsis
getselectcallback()

Returns
A function or 0

Description

This function returns the “select callback” function (set with the function setselectcallback). It is part of the communications between the Regionmanager and the `region-of-interest` entry widget. If you have not set a callback function, this function returns 0.
It is not intended for general purpose use and you will break some communications if you mess with these functions.

---

regionmanager.gui - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Start the GUI interface

Synopsis
gui(parent, tlead, tpos)

Arguments

parent	in	Parent of the widget Allowed:Tk frame Default: F
tlead	in	Lead frame Allowed:Tk frame Default: F
tpos	in	Position of GUI relative to lead Allowed:String Default: 'sw'

Returns
Bool or fail

Description

This function starts the GUI interface for the regionmanager. You can embed the GUI in a parent frame if desired. For world regions in particular, the GUI interface is better than the command line interface. Popuphelp is available on many of the buttons, labels and list box entries. In general, look for popuphelp on labels of entry boxes rather than the entry boxes themselves.
The `tlead, tpos` arguments are the same as for the Glish/Tk frame command and control where the GUI goes. By default it goes where it feels like. If you dismiss the GUI and reissue the `gui` command, it is just remapped, so the arguments are ignored. If you destroy the GUI (under the File button) then the arguments for the `gui` command are honoured.
Here is an example of how the GUI might look.
The main control panel of the GUI shows three list boxes. To the left is a list of the different types of regions that you can create. In the middle is a list of image `tools` currently available to *Glish* (two in this example). To the right is the list of regions that have been made so far (in this example, some pixel, world and compound regions).
Just click on one of the region types in the left hand list box and an additional GUI will appear for that region type. Enter the parameters for the region and press the 'create' button of the specific region GUI. The new region will then appear in the right-hand list box of the main control panel.
For world regions, you must select an image from the middle list box first, and then click on the region type you wish to create.
Other buttons on the main GUI function as follows.
1. For the 'Images' listbox. The 'deselect' button deselects all items in the images listbox.
2. For the 'Available Regions' listbox. The 'Delete' button deletes any selected regions. The 'Undelete' button can be used to recover the region last deleted. The 'Edit' button lets you edit a selected region. The 'Copy' button lets you copy one selected region to a new region. The 'Rename' button lets you

rename one selected region. The 'Clipboard' button is in fact a menu from which you can copy the selected region to the default clipboard tool (**dcb**) or paste a region from **dcb** to *Glish*. For the latter operation, the region name will be automatically chosen.
3. Along the bottom of the GUI we find the 'Refresh' button to refresh/update the 'Images' and 'Available Regions' listboxes with images and regions currently available to *Glish*. The 'Dismiss' button dismisses the GUI but preserves its state. You can recover it but issuing the `gui` function call again. The other three buttons are used in the communications with region entry widgets used for example, by the Toolmanager interface. What happens is that you can initiate, from the region entry widget, a connection to the Regionmanager (by selecting the RegionmanagerGUI entry from the spanner menu). Once you have done that, the **drm** Regionmanager GUI will be displayed and the 'Send&dismiss', 'Send' and 'Break' buttons will be enabled. You then select the desired region in the 'Available Regions' listbox. You then send it to the region entry widget by pressing either the 'Send&dismiss' or 'Send' buttons. When you press the former, the Regionmanager GUI is dismissed and the connection broken. When you press the latter, the Regionmanager GUI remains and the connection persists. If you want to just break the connection, press the 'Break' button (these three buttons then disable).
Note that only one region entry widget at a time can be connected to the **drm** Regionmanager GUI (otherwise you might get a bit confused about which regions are going where).
4. The 'File' menu button at the top left of the GUI gives you 5 choices. The first three enable you to save regions to Table (e.g. an image), restore then from a Table and delete regions from a Table, respectively. Each of these starts its own GUI for these actions. These are described in saveregions, restoreregions, and deleteregions, respectively.
Regarding their interaction with the Regionmanager, the following is relevant. With the 'Save' widget, the regions that you have selected in the Regionmanager 'Available Regions' listbox are available for saving. Whenever you change the Regionmanager selection, the Save widget updates.
With the 'Restore' widget, the regions available for restoring to *Glish* come from the first selected image in the Regionmanager GUI 'Images' listbox (or from the Table you type in to the Restore widget).
With the 'Delete' widget, the Table that you are deleting regions from comes from the first selected image in the Regionmanager GUI 'Images' listbox (or from the Table you type in to the Restore widget).
The other two items in the 'File' menu dismiss the Regionmanager GUI (preserving its state) and destroy the Regionmanager GUI. In both cases the GUI can be recovered by running the `gui` function of the Regionmanager again.

Example

```
- drm.gui()
T
```

---

regionmanager.intersection - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create the intersection of some world regions

Synopsis
intersection(...)

Arguments

...	in	World regions and comment Allowed:Region tools or record of region tools, and String

Returns
Region tool

Description

This function (short-hand name `int`) creates a region which is the intersection of the given world regions. The input regions can themselves be compound regions (such as the union or intersection etc). The input regions can be provided as individual region `tools`, or a record of region `tools` (see examples).
For arcane *Glish* language reasons, the usual `comment` keyword is not provided. But if you provide a string somewhere in the call sequence, it will be treated as the comment.

Example

```
- im := image('hcn')
- cs := im.coordsys()
- drm.setcoordinates(cs)
-
- blc := "10pix 10pix 1pix"
- trc := "60pix 60pix 1pix"
- r1 := drm.wbox(blc=blc, trc=trc, pixelaxes=[1,2,3])
-
- x := dq.quantity([50,55,58,65,58,53,50], 'pix')
- y := dq.quantity([50,53,69,70,63,58,55], 'pix')
- r2 := drm.wpolygon(x=x, y=y, pixelaxes=[1,2])
-
- r3 := drm.intersection(r1, r2, 'This is the comment')
-
- im.boundingbox(r1)
[blc=[10 10 1] , trc=[60 60 256] , regionShape=[51 51 256] , imageShape=[155 178 256] ]
- im.boundingbox(r2)
[blc=[50 50 1] , trc=[65 70 256] , regionShape=[16 21 256] , imageShape=[155 178 256] ]
- im.boundingbox(r3)
[blc=[50 50 1] , trc=[60 60 256] , regionShape=[11 11 256] , imageShape=[155 178 256] ]
-
- im.stats(region=r3)                          # Some output discarded
NORMAL: Selected bounding box [50, 50, 1] to [60, 60, 1]
Number points = 51
```

In this example, we use pixel coordinates so that it is clear what is happening. You can see that the number of pixels in the intersection (51) is less than the number in the bounding box of the intersection (121) because the intersection is actually polygonal and does not fill the bounding box.

Example

```
- im := image('onno')
- cs := im.coordsys()
- x := dq.quantity([3,6,9,6,5,5,3],'pix')
- y := dq.quantity([3,4,7,9,7,5,5],'pix')
-
- regions := [=];
- regions[1] := drm.wpoly(x,y,[1,2],cs)
-
- blc := "17:42:29.303 -28.59.18.600"
- trc := "17:42:28.303 -28.59.10.600"
- regions[2] := drm.wbox(blc,trc,[1,2],cs)
-
```

```
- r3 := drm.intersection(regions,'The mysteries of Glish')
```

This example is the same as the prevoius one, except the regions are provided to the intersection function in a record, rather than directly in the call sequence.

---

regionmanager.ispixelregion - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Is this region a pixel region ?

Synopsis
ispixelregion(region)

Arguments

region	in	The region Allowed:Region tool

Returns
Bool

Description

This function returns T if the region is a pixel region. For any other *Glish* variable it returns F.

Example

```
- im := image('hcn')
- cs := im.coordsys()
- r1 := drm.box()                  # A pixel region
- r2 := drm.wbox(csys=cs)          # A world region
- drm.ispixelregion(r1)
T
- drm.ispixelregion(r2)
```

```
F
- x := [20,30]
- drm.ispixelregion(x)
F
```

---

regionmanager.isworldregion - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Is this region a world region ?

Synopsis
isworldregion(region)

Arguments

region	in	The region Allowed:Region tool

Returns
Bool

Description

This function returns T if the region is a world region. For any other *Glish* variable it returns F.

Example

```
- im := image('hcn')
- cs := im.coordsys()
- r1 := drm.box()                  # A pixel region
- r2 := drm.wbox(csys=cs)          # A world region
- drm.isworldregion(r1)
F
- drm.isworldregion(r2)
```

```
T
- x := [20,30]
- drm.isworldregion(x)
F
```

---

regionmanager.namesintable - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Find the names of the regions stored in a Table

Synopsis
namesintable(tablename)

Arguments

tablename	in	The table Allowed:Image tool, table tool or String

Returns
Vector of strings

Description

This function returns the names of regions stored in an AIPS++ Table.
For the `tablename` argument, you can specify an image tool, a table tool, or a string. If you give a string, it should be the name of an existing AIPS++ table on disk (any kind of table).

Example

```
- im := image('hcn')
- names := drm.namesintable(im)
- names
r1 poly2 int0
```

---

regionmanager.pixeltoworldregion - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Convert a simple pixel region to a world region

Synopsis
pixeltoworldregion(csys, shape, region)

Arguments

csys	in	The coordinate system to use in the conversion Allowed:Image tool
shape	in	The shape to use for fractional coordinates Allowed:Vector of integers Default: Unset
region	in	The region Allowed:Region tool

Returns
Region tool or fail

Description

This function converts the simple pixel regions to world regions. This function is provided because compound regions require world regions. You specify a Coordinate System `tool` for the conversions as well as the pixel region. You only need to supply a shape if the coordinates use fractional coordinates. Currently, this function can only convert pixel boxes.

Example

```
- im := image('hcn')
- cs := im.coordsys()
- r1 := drm.box([10,20,30], [50,60,70])
- r2 := drm.pixeltoworldregion(csys=cs, region=r1)
- drm.isworldregion(r1)
F
```

```
- drm.isworldregion(r2)
T
- im.boundingbox(r1)
[blc=[10 20 30] , trc=[50 60 70] , inc=[1 1 1] , bbShape=[41 41 41] ,
regionShape=[41 41 41] , imageShape=[155 178 256] ]
-
- im.boundingbox(r2)
[blc=[10 20 30] , trc=[50 60 70] , inc=[1 1 1] , bbShape=[41 41 41] ,
regionShape=[41 41 41] , imageShape=[155 178 256] ]
```

---

regionmanager.pseudotoworldregion - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Convert a Viewer pseudoregion to a world region

Synopsis
pseudotoworldregion(image, pseudoregionvalue, ddoptions, intersect)

Arguments

image	in	Image of interest Allowed:Image tool
pseudoregionvalue	in	Psuedoregion event value Allowed:Glish record
ddoptions	in	Options from display data Allowed:Record
intersect	in	Intersect with move plane ? Allowed:T or F Default: T

Returns
Region tool or fail

Description

This function is for use with the `pseudoregion` event emitted by a Displaypanel created by a Viewer. This function converts the pseudo-region (a

region of 2-dimensions only) into a true world region by application of the Coordinate System. If the record is empty, the returned region describes the entire image.
This function is not intended for general-user use.
You must provide

- `image` This is the Image `tool` that the conversion pertains to.
- `pseudoregionvalue` is the $value of the *Glish* `pseudoregion` event emitted by a Displaypanel (created by a Viewer).
- `ddoptions` is a record obtained from the display data `tool` via the getoptions function. It contains information about what is displayed.

If `intersect=T` then the pseudoregion is intersected with the plane of the current movie axis and hidden axes. If `intersect=F` then the pseudoregion is intersected with the current hidden axes pixels only.

Example

```
include 'image.g'
include 'viewer.g'
include 'regionmanager.g'
#
im : =image('myimage')
displaypanel := dv.newdisplaypanel()
displaydata := dv.loaddata(im.name(F), 'raster')
r1 := drm.quarter()
displaydata.setoptions([region=r1])
displaypanel.register(displaydata)
#
whenever displaypanel->pseudoregion do {
  options := displaydata.getoptions()
  region := drm.pseudotoworldregion(im, $value, options)
  if (is_fail(region)) fail
  im.statistics(region=region)
}
```

In this example, we use the default viewer (`dv`) to create a `displaypanel` on which we register a `displaydata` which will display the specified image as a raster. A region specifying a quarter of the image is given. When the user interactively generates a region, the event is caught and the pseudo-region event converted to a true world region with the default regionmanbager, `drm`. The region is then used to evaluate statistics from the image.

---

regionmanager.quarter - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create a quarter pixel region

Synopsis
quarter(comment)

Arguments

comment	in	A comment stored with the region Allowed:String

Returns
Region tool or fail

Description

This function creates a region which selects one quarter of the image by area centered on the image center for the first two dimensions. It selects all pixels for higher dimensions. It might be useful for deconvolution (CLEAN). This region is implemented with the pixel box function.

Example

```
- r1 := drm.quarter()
- im := image('hcn')
- im.boundingbox(r1)
[blc=[40 46 1] , trc=[116 134 256] , inc=[1 1 1] , bbShape=[77 89 256] ,
regionShape=[77 89 256] , imageShape=[155 178 256] ]
```

You can see that auto extension to the third axis is occurring.

---

regionmanager.setcoordinates - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Set new default Coordinate System

Synopsis
setcoordinates(csys, verbose)

Arguments

csys	in	Default Coordinate System for use in world regions Allowed:Coordinate tool
verbose	in	Issue message each time used Allowed:T or F Default: T

Returns
Bool

Description

This function allows you to (re)set the default Coordinate System used by the functions that make world regions. If you don't specifiy a Coordinate System when you make the world region, the default Coordinate System, if there is one, is used. The Coordinate System is stored in a **coordinates** `tool` and is created with the coordsys `tool function`.
Normally, the world region creating functions like wbox and wpolygon will issue a message each time the private Coordinate System is used. However, if you set `verbose=F` then this will not occur.

Example

```
- im := image('quiqui')
- cs := im.coordsys()
- drm.setcoordinates(cs)
- r1 := drm.wbox()
Using private CoordinateSystem from image "quiqui"
```

---

regionmanager.setselectcallback - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Set the callback function when GUI region selection is made

Synopsis
setselectcallback(callbackFunction)

Arguments

callbackFunction	in	The function Allowed:A function

Returns
T

Description

This function sets the select callback function (retrievable with the function getselectcallback). It is part of the communications between the Regionmanager and the `region-of-interest` entry widget.
It is not intended for general purpose use and you will break some communications if you mess with these functions.

---

regionmanager.type - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Return the type of this tool

Synopsis
type()

Returns
String

Description

This function returns the string 'regionmanager'. If you have a `tool`, but don't know what kind it is, this function usually exists to tell you that.

Example

```
- if (is_defined('thing') && is_record(thing) &&
+     has_field(thing, 'type') && is_function(thing.type)
+     && thing.type() == 'regionmanager') {
+    print 'yippee'
+ } else {
+    print 'it is a sad day that we do not have a regionmanager'
+ };
```

Imagine we have been given a *Glish* variable called `thing` in some function but we can't guarentee its a Regionmanager, as expected. To write robust code, we need to check that it is. This bit of code does that. Note the use of the short cut "&&" operator. A statement is only evaluated if the statement to the left is true.

---

regionmanager.union - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create a union of world regions

Synopsis

union(...)

Arguments

...	in	World regions and comment	
		Allowed:	Region tools or record of region tools, and String

Returns

Region tool

Description

This function takes a minimum of two world regions and creates a region which is the union of the given regions. The input regions can themselves be compound regions (such as the union or intersection etc). The input regions can be provided as individual region `tools`, or a record of region `tools` (see examples).
For arcane *Glish* language reasons, the usual `comment` keyword is not provided. But if you provide a string somewhere in the call sequence, it will be treated as the comment.

Example

```
- im := image('onno')
- cs := im.coordsys()
- x := dq.quantity([3,6,9,6,5,5,3],'pix')
- y := dq.quantity([3,4,7,9,7,5,5],'pix')
- r1 := drm.wpoly(x,y,[1,2],cs)
-
- blc := "17:42:29.303 -28.59.18.600"
- trc := "17:42:28.303 -28.59.10.600"
- r2 := drm.wbox(blc,trc,[1,2],cs)
-
- r3 := drm.union(r1,r2,'The mysteries of Glish')
-
- im.shape()
[155 178 256]
- im.boundingbox(r1)
[blc=[3 3 1] , trc=[9 9 256] , inc=[1 1 1] , bbShape=[7 7 256] ,
regionShape=[7 7 256] , imageShape=[155 178 256] ]
- im.boundingbox(r2)
[blc=[90 90 1] , trc=[103 98 256] ,  inc=[1 1 1] , bbShape=[14 9 256] ,
regionShape=[14 9 256] , imageShape=[155 178 256] ]
- im.boundingbox(r3)
[blc=[3 3 1] , trc=[103 98 256] ,  inc=[1 1 1] , bbShape=[101 96 256] ,
regionShape=[101 96 256] , imageShape=[155 178 256] ]
-
- im.stats(region=r1)
Selected bounding box [3, 3, 1] to [9, 9, 256]
Number points = 6400
-
- im.stats(region=r2)
Selected bounding box [90, 90, 1] to [103, 98, 256]
Number points = 32256
-
- im.stats(region=r3)
```

```
Selected bounding box [3, 3, 1] to [103, 98, 256]
Number points = 38656
```

When the polygon only is applied, it is auto extended along the third axis. The `statistics` function finds 6400 pixels in the region, which is $6400/256 = 25$ pixels per plane. Likewise, when the box only is applied, the `statistics` function finds 32256 pixels in the region, which is $32256/256 = 126$ pixels per plane. When the union is applied, the `statistics` function finds 38656 pixels in the region. First it finds the union of the polygon and box (which are specified only in the XY plane) and that union is extended. Thus we expect $(25 + 126) * 256 = 38656$ pixels in the region of the union, as found.

Example

```
- im := image('onno')
- cs := im.coordsys()
- x := dq.quantity([3,6,9,6,5,5,3],'pix')
- y := dq.quantity([3,4,7,9,7,5,5],'pix')
-
- regions := [=];
- regions[1] := drm.wpoly(x,y,[1,2],cs)
-
- blc := "17:42:29.303 -28.59.18.600"
- trc := "17:42:28.303 -28.59.10.600"
- regions[2] := drm.wbox(blc,trc,[1,2],cs)
-
- r3 := drm.union(regions,'The mysteries of Glish')
```

This example is the same as the prevoius one, except the regions are provided to the union function in a record, rather than directly in the call sequence.

---

regionmanager.wbox - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create a world box region

Synopsis

wbox(blc, trc, pixelaxes, csys, absrel, comment)

Arguments

blc	in	blc of the box Allowed:Vector of quantities Default: Unity
trc	in	trc of the box Allowed:Vector of quantities Default: Shape
pixelaxes	in	Which pixel axes Allowed:Vector of integers Default: [1,2,...]
csys	in	Coordinate System Allowed:coordinates tool Default: Private Coordinate System
absrel	in	Absolute or relative coordinates Allowed:Vector of strings from 'abs', 'relref' and 'relcen' Default: 'abs' or last
comment	in	A comment stored with the region Allowed:String

Returns

Region tool

Description

This function creates a multi-dimensional world box region; the corners of the box are specified in world coordinates. However, the box is not a true world volume in that its sides do not follow world contours. Its sides are parallel to the pixel axes. If you are in a region of high world coordinate contour non-linearity (e.g. near a pole), you are probably better off using a world polygon.

The box is specified by a bottom-left corner, and a top-right corner. The coordinates are given as quantities, and you can give a vector of quantities (e.g. `blc := dq.quantity("1rad 20deg")` or a quantity of a vector (e.g.`blc := dq.quantity([10,30], 'rad')`).

You can specify whether the coordinates are given as absolute coordinates (`absrel='abs'`) or relative to the reference pixel (`absrel='relref'`) or relative to the center of the image (`absrel='relcen'`). You can specify this for each axis (the same for the blc and trc). If you specify less values than the number of values in `blc` or `trc` then the last value you did specify is used as the default for all higher numbered axes (e.g. `absrel='relref'` means `absrel="relref relref"` for two axes).

You specify which pixel axes in the image the `blc` and `trc` vector refer to with the `pixelaxes` argument. If you don't, it defaults to [1,2,...]. This

specification is an important part of world regions.
You must also specify the Coordinate System with the `csys` argument. The Coordinate System is encapsulated in a `coordinates` tool and can be recovered from an image with the coordsys tool function. You can also set a default Coordinate System in the Regionmanager with the setcoordinates function.
In the Regionmanager we have defined units 'pix' and 'frac'; these are then known to the quanta system. This means that you can effectively define a pixel box (except for the stride capability) as a world box with most of the advantages of world regions (can be used for compound regions). However, it is still not very portable to other images because the coordinates are pixel based, not world based.
Note that the need to deal with the `pixelaxes` and `csys` is hidden from you when using the gui interface of the Regionmanager.

Example

```
- r := drm.wbox()
```

This region, on application to an image, will select the entire image.

Example

```
- im := image('ada')
- cs := im.coordsys()
- cs.summary()

Name             Proj Shape Tile   Coord value at pixel    Coord incr Units
---------------------------------------------------------------------------
Frequency              64   16  1.413350e+09     1.00  1.968717e+04 Hz
Velocity                        1.378053e+02     1.00 -4.174021e+00 km/s
Declination       SIN   178   89 -28.59.18.600    90.00  1.000000e+00 arcsec
Right Ascension   SIN   155   31  17:42:29.303    90.00 -1.000000e+00 arcsec
-
-
- blc := dq.quantity("17:42:29.303 -28.59.18.600")
- trc := "17:42:28.303 -28.59.10.600"
- r1 := drm.wbox(blc=blc,trc=trc,pixelaxes=[3,2],csys=cs)
- im.boundingbox(r1)
[blc=[1 90 90] , trc=[64 98 103] , regionShape=[64 9 14], imageShape=[64 178 155] ]
```

We have specified an RA and DEC for the blc and the trc (they should be quantities; for blc we do that explicitly, but for the trc we just give a vector of strings which is automatically converted for us to a vector of quantities).

From the summary listing you can see that RA and DEC correspond to pixel axes 3 and 2 respectively (don't be confused by the dual listing for the spectral axis) so that is why the pixelaxes argument is set to [3,2]. If we had set blc/trc in DEC/RA order then we would have put pixelaxes=[2,3]. For the unspecified frequency axis, all pixels are selected.

Example

```
- im := image('bork')
- cs := im.coordsys()
- cs.summary()

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   89 -28.59.18.600    90.00  1.000000e+00 arcsec

- drm.setcoordinates(cs)
T
- blc := "-10pix -28.59.18.6"
- trc := "10pix  -28.59.1.6"
- r1 := drm.wbox(blc=blc,trc=trc,absrel="relref abs")   # pixelaxes defaults to [1,2]
Using private CoordinateSystem from image "bork"
- im.boundingbox(r1)
[blc=[80 90] , trc=[100 107] , regionShape=[21 18] , imageShape=[155 178] ]
```

In this example, we use pixel coordinates relative to the reference pixel for the RA axis and absolute world coordinates for the DEC axis. We also set the state of the Regionmanager with a Coordinate System to use when making world regions. You can see that when the region was made, a message was issued reminding you that the internal Coordinate System from the image *bork* was being used.

Example

```
- im := image('hcn')
- cs := im.coordsys()
- cs.summary()

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   89 -28.59.18.600    90.00  1.000000e+00 arcsec
```

```
Frequency                64   16  1.413350e+09     1.00  1.968717e+04 Hz
Velocity                          1.378053e+02     1.00 -4.174021e+00 km/s
T
-
- blc := "1.414E9Hz"
- trc := "1.4145GHz"
- r := drm.wbox(blc=blc, trc=trc, pixelaxes=[3], csys=cs)
- im.boundingbox(r)
[blc=[1 1 34] , trc=[155 178 59] , regionShape=[155 178 26] , imageShape=[155 178 64] ]
```

In this example we only specified a region for the frequency axis (note we used different units for the blc and trc). Therefore, on application, the region selected for the RA and DEC axes is automatically the full image.

---

regionmanager.wpolygon - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create a world polygon region with quantities

Synopsis

wpolygon(x, y, pixelaxes, csys, absrel, comment)

Arguments

x	in	The x vector Allowed:Quantity vector
y	in	The y vector Allowed:Quantity vector
pixelaxes	in	which pixel axes Allowed:Vector of integers or strings Default: [1,2]
csys	in	Coordinate System Allowed:coordinates tool Default: Privaet Coordinate System
absrel	in	Absolute or relative coordinates Allowed:String from 'abs', 'relref' and 'relcen' Default: 'abs'
comment	in	A comment stored with the region Allowed:String

Returns
Region tool

Description

This function (short-hand name `wpoly`) creates a 2D world polygon region. The polygon is specified by an `x` and a `y` vector. These must be quantities of a vector (the world box function allows both quantities of vectors and vectors of quantities). This means that the units are common to all elements of each vector. Thus, `dq.quantity([1,2,3],'rad')` (a quantity of a vector) is different from `dq.quantity("1rad 2rad 3rad")` (a vector of quantities) although the information that they specify is the same.
You specify which pixel axes in the image the `x` and `y` vectors pertain to with the `pixelaxes` argument. If you don't, it defaults to [1,2]. This specification is an important part of world regions.
You can specify whether the `x` and `y` vector coordinates are given as absolute coordinates (`absrel='abs'`) or relative to the reference pixel (`absrel='relref'`) or relative to the center of the image (`absrel='relcen'`). This argument applies to both the axes of the polygon.
You must also specify the Coordinate System with the `csys` argument. The Coordinate System is encapsulated in a `coordinates tool` and can be recovered from an image with the coordsys function. You can also set a default Coordinate System in the Regionmanager with the setcoordinates function.
In the Regionmanager we have defined units 'pix' and 'frac'; these are then known to the quanta system. This means that you can effectively define a pixel box (except for the stride capability) as a world box with most of the advantages of world regions (can be used for compound regions). However, it is still not very portable to other images because the coordinates are pixel based, not world based.
Note that the need to deal with the `pixelaxes` and `csys` is hidden from you when using the gui interface of the Regionmanager.

Example

```
- im := image('onno')
- cs := im.coordsys()
- im.shape()
[155 178 64]
- x := dq.quantity([3,6,9,6,5,5,3],'pix')
- y := dq.quantity([3,4,7,9,7,5,5],'pix')
- r1 := drm.wpolygon(x=x, y=y, pixelaxes=[2,3], csys=cs)
- im.statistics(region=r1)
Selected bounding box [1, 3, 3] to [155, 9, 9]
```

```
Creating new statistics storage image

Number points = 3875
Sum           = -9.457746e+01        Mean     = -2.440709e-02
Variance      = 2.599452e-02         Sigma    = 1.612282e-01
Rms           = 1.630445e-01

Minimum value at [50, 7, 5] = -6.885301e-01
Maximum value at [155, 3, 5] = 6.562161e-01
```

We applied the 2D polygon, defined in the YZ plane with absolute pixel coordinates, to a 3D image. Therefore, the first (X) axis was automatically extended to the whole image.

---

regionmanager.wmask - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create a world mask region

Synopsis
wmask(expr)

Arguments

expr	in	Boolean expression Allowed:String

Returns
Region tool

Description

This function creates a mask region. This region can be applied on-the-fly to images. The expression must be Boolean.

Example

```
- im := imagemaketestimage('zz')
- r := drm.wmask(expr='zz>0')
- im.statistics(region=r)
```

The mask region effectively creates a (notional) Boolean lattice. In this example, when the pixel values are greater than zero, the lattice is True (good), else False. The statistics, when evaluated, are then only computed when the mask region is True.

Example

```
- im := imagemaketestimage('zz')
- drm.setcoordinates(im.coordsys())
#
- r1 := drm.wbox(...)                 # Make a world box
- r2 := drm.wmask(expr='zz>0')        # Make a mask region
- r3 := drm.intersection (r1,r2)      # Intersect them
- im.statistics(region=r3)            # Find statistics
```

In this example, the mask is only applied when the pixels fall in the specified box.

---

regionmanager.wrange - Function

Package	**general**
Module	**images**
Tool	**regionmanager**

Create a world region for a pixel range on one axis

Synopsis

wrange(range, pixelaxis, csys, absrel, comment)

Arguments

range	in	World range Allowed:Numeric or quantum vector
pixelaxis	in	The pixel axis of interest Allowed:Integer
csys	in	Coordinate System Allowed:coordinates tool Default: Private Coordinate System
absrel	in	Absolute or relative coordinates Allowed:Vector of strings from 'abs', 'relref' and 'relcen' Default: 'abs' or last
comment	in	A comment stored with the region Allowed:String

Returns
Region tool or fail

Description

This function creates a region which selects all pixels except for the axis specified. For that axis it selects just the pixels specified by the range. The region created is just a world box.

Example

```
- im := image('hcn')
- cs := im.coordsys();
- drm.setcoordinates(cs);
-
- im.toworld([1,1,10])
[4.63647352 -0.506376202 1.41352721e+09]
- im.toworld([1,1,20])
[4.63647352 -0.506376202 1.41372408e+09]
-
- r := drm.wrange(range="1.41352721MHz 1.4137240MHz", pixelaxis=3, csys=cs)
- im.boundingbox(r)
[blc=[1 1 10] , trc=[155 178 20] , inc=[1 1 1] , bbShape=[155 178 11] ,
 regionShape=[155 178 11] , imageShape=[155 178 64] ]
```

To show that it works, we converted from pixel to world coordinate for pixel 10 and 20 on the third axis. These values, with a unit change to MHz from GHz were then input into the `wrange` function and the bounding box shows that we recover 10 and 20 pixels.

Example

```
- im := image('hcn')
- cs := im.coordsys();
- drm.setcoordinates(cs);
- r := drm.wrange(range="10pix 20pix", pixelaxis=3, csys=cs)
- im.boundingbox(r)
[blc=[1 1 10] , trc=[155 178 20] , inc=[1 1 1] , bbShape=[155 178 11] ,
 regionShape=[155 178 11] , imageShape=[155 178 64] ]
-
- r := drm.wrange(range=[10, 20], pixelaxis=3, csys=cs)
- im.boundingbox(r)
[blc=[1 1 10] , trc=[155 178 20] , inc=[1 1 1] , bbShape=[155 178 11] ,
 regionShape=[155 178 11] , imageShape=[155 178 64] ]
-
```

This shows how you can easily select a few planes of an image using pixel units. If you give a numeric vector for the range (second example) it assumes pixel units.

1.2.9 is_region - Function

Package general
Module images

Is this a valid region tool?

Synopsis
is_region(thing)

Arguments

thing	in	The thing to test Allowed:Any

Returns
Bool

Description

Determine if a given *Glish* variable is a **region** `tool`.

Example

```
- r1 := drm.box()
- is_region(r1)
T
- r2 := [10,20]
- is_region(r2)
F
```

1.2.10 coordsys - Tool

Package general
Module images

Operations on CoordinateSystems
include "coordsys.g"

Constructors

coordsys	Create a default coordsys tool

Functions

axesmap	Find mapping between world and pixel axes
axiscoordinatetypes	Return types of coordinates for each axis
conversiontype	Get extra reference conversion layer
convert	Convert a numeric mixed coordinate
convertmany	Convert many numeric mixed coordinates
coordinatetype	Return type of specified coordinate
copy	Copy this Coordsys tool
done	Destroy this Coordsys tool
epoch	Return the epoch
findaxis	Find specified axis in coordinate system
findcoordinate	Find axes of specified coordinate
frequencytofrequency	Convert frequency to frequency with an offset
frequencytovelocity	Convert frequency to velocity
fromrecord	Fill Coordinate System from a record
id	Return the fundamental identifier of this tool
increment	Recover the increments

lineartransform	Recover the linear transform matrix
names	Recover the names for each axis
naxes	Recover the number of axes
ncoordinates	Recover the number of coordinates in the Coordinate System
observer	Return the name of the observer
projection	Recover the direction coordinate projection
referencecode	Return specified reference code
referencepixel	Recover the reference pixel
referencevalue	Recover the reference value
reorder	Reorder the coordinates
replace	Replace a coordinate
restfrequency	Recover the rest frequency
setconversiontype	Set extra reference conversion layer
setdirection	Set direction coordinate values
setepoch	Set a new epoch
setincrement	Set the increment
setlineartransform	Set the linear transform
setnames	Set the axis names
setobserver	Set a new observer
setprojection	Set the direction coordinate projection
setreferencecode	Set new reference code
setreferencelocation	Set reference pixel and value
setreferencepixel	Set the reference pixel
setreferencevalue	Set the reference value
setrestfrequency	Set the rest frequency
setspectral	Set tabular values for the spectral coordinate
setstokes	Set the Stokes types
settabular	Set tabular values for the tabular coordinate
settelescope	Set a new telescope
setunits	Set the axis units
stokes	Recover the Stokes types
summary	Summarize basic information about the Coordinate System
telescope	Return the telescope
toabs	Convert relative coordinate to absolute
toabsmany	Convert many numeric relative coordinates to absolute
topixel	Convert from absolute world to pixel coordinate
topixelmany	Convert many absolute numeric world coordinates to pixel
torecord	Convert Coordinate System to a record
torel	Convert absolute coordinate to relative
torelmany	Convert many numeric absolute coordinates to relative
toworld	Convert from absolute pixel coordinate to world
toworldmany	Convert many absolute pixel coordinates to numeric world
type	Return the type of this tool
units	Recover the units for each axis
velocitytofrequency	Convert velocity to frequency

Summary

A Coordsys `tool` is used to store and manipulate a Coordinate System (we will use the term 'Coordinate System' interchangeably with 'Coordsys `tool`'). A Coordinate System is a collection of coordinates, such as a direction coordinate (E.g. RA/DEC), or a spectral coordinate (e.g. an LSRK frequency).
The main job of the Coordsys `tool` is to convert between absolute pixel and world (physical) coordinates. It also supports relative pixel and world coordinates (relative to reference location).
A Coordinate System is generally associated with an image (manipulated via an Image `tool`) but can also exist in its own right. An image is basically just a regular lattice of pixels plus a Coordinate System describing the mapping from pixel coordinate to world (or physical) coordinate.
Each coordinate is associated with a number of axes. For example, a direction coordinate has two coupled axes; a longitude and a latitude. A spectral coordinate has one axis. A linear coordinate can have an arbitrary number of axes, but they are all uncoupled. The Coordinate System actually maintains two kinds of axes; pixel axes and world axes.
As well as the coordinates, there is some extra information stored in the Coordinate System. This consists of the telescope, the epoch (date of observation), and the highly influential observer's name. The telescope (i.e. position on earth) and epoch are important if you want to, say, change a spectral coordinate from LSRK to TOPO.
For general discussion about celestial coordinate systems, see the papers by Mark Calabretta and Eric Greisen. Background on the WCS system and relevant papers (including the papers published in

```
A&A 2002, 1061-1075 and 1077-1122
```

can be found here. Note that the actual system implemented originally in AIPS++ was based on a 1996 draft of these papers. The final papers are being implemented while new version of the defining library become available.

Coordinate formatting

Many of the Coordsys `tool` functions use a world coordinate value as an argument. This world value can be formatted in many ways.
Some functions (e.g. toworld) have a function argument called `format` which takes a string. This controls the format in which the coordinate is output and hence possibly input into some other function.
Possibilities for `format` are :

- 'n' - means the world coordinate is given as a numeric vector (actually doubles). The units are implicitly those returned by function units.
- 'q' - means the world coordinate is given as a vector of quantities (value and unit) - see the quanta module. If there is only one axis (e.g. spectral coordinate), you will get a single quantum only.

- 'm' - means the world coordinate is given as a record of measures - see the measures module.

  The record consists of fields named from from `direction`, `spectral`, `stokes`, `linear`, and `tabular`, depending upon which coordinate types are present in the Coordinate System.

  The `direction` field holds a direction measure.

  The `spectral` field holds further subfields `frequency`, `radiovelocity`, `opticalvelocity`, `betavelocity`. The `frequency` subfield holds an frequency measure. The `radiovelocity` subfield holds a doppler measure using the radio velocity definition. The `opticalvelocity` subfield holds a doppler measure using the optical velocity definition. The `betavelocity` subfield holds a doppler measure using the true or beta velocity definition.

  The `stokes` field just holds a string giving the Stokes type (not a real measure).

  The `linear` and `tabular` fields hold a vector of quanta (not a real measure).

- 's' - means the the world coordinate is given as a vector of formatted strings

You can give any combination of the allowed letters when using the `format` argument. If you supply one letter the coordinate is as described above. If you supply more than one letter, the coordinate is given as a record, with possible fields 'numeric', 'quantity', 'measure' and 'string' where each of these fields is given as described above.

There are functions torel and toabs used to inter-convert between absolute and relative world and pixel coordinates. These functions have an argument `isworld` whereby you can specify whether the coordinate is a pixel coordinate or a world coordinate. In general, you should not need to use this argument because any coordinate variable generated by Coordsys `tool` functions 'knows' whether it is absolute or relative, world or pixel (uses a *Glish* attribute). However, you may be inputting a coordinate variable which you have generated in some other way, and then you may need this.

Stokes Coordinates
Stokes axes don't fit very well into our Coordinate model since they are not interpolatable. The alternative to having a Stokes Coordinate is having a Stokes pixel type (like double, complex). Both have their good and bad points. We have chosen to use a Stokes Coordinate.
With the Stokes Coordinate, any absolute pixel coordinate value must be in the range of 1 to $nStokes$, where $nStokes$ is the number of Stokes types in the Coordinate.

We define relative world coordinates for a Stokes axis to be the same as absolute world coordinates (it makes no sense to think of a relative value $XY - XX$ say).
You can use the specialized functions stokes and setstokes to recover and set new Stokes values in the Stokes Coordinate.

World and Pixel axes
The Coordinate System maintains what it calls pixel and world axes. The pixel axis is associated with, say, the axes of a lattice of pixels. The world axes describe notional world axes, generally in the same order as the pixel axes. However, they may be different. Imagine that a 3-D image is collapsed along one axis. The resultant image has 2 pixel axes. However, we can maintain the world coordinate for the collapsed axis (so we know the coordinate value still). Thus we have three world axes and two pixel axes. It is also possible for the C++ programmer to reorder these pixel and world axes. However, this is strongly discouraged, and you should never actually encounter a situation where the pixel and world axes are in different orders, but you may encounter cases where the number of world and pixela axes is different.
For those of us (AIPS++ programmers) writing robust scripts, we must account for these possibilities, although the user really shouldn't bother. This, the `pixel` and `world` vectors return the pixel and world axes of the found coordinate.
The functions referencevalue, increment, units, and names return their vectors in world axis order. However, function referencepixel returns in pixel axis order (and the world vectors might have more values than the referencepixel vector).

Overview of Coordsys tool functions

- **Get/set -** Functions to get and set various items within the Coordinate System are
  - referencepixel - get the reference pixel
  - setreferencepixel - set the reference pixel
  - referencevalue - get the reference value
  - setreferencevalue - set the reference value
  - setreferencelocation - Set reference pixel and value to these values
  - increment - get axis increments
  - setincrement - set axis increments
  - lineartransform - get linear transform
  - setlineartransform - set linear transform
  - names - get axis names
  - setnames - set axis names
  - units - get axis units

- setunits - set axis units
- stokes - get Stokes values
- setdirection - set Direction coordinate values
- setstokes - set Stokes values
- setspectral - set Spectral coordinate tabular values
- settabular - set Tabular coordinate tabular values
- projection - get direction coordinate projection
- setprojection - set direction coordinate projection
- referencecode - get reference codes
- setreferencecode - set reference codes
- restfrequency - get the spectral coordinate rest frequency
- setrestfrequency - set the spectral coordinate rest frequency
- epoch - get the epoch of observation
- setepoch - set the epoch of observation
- telescope - get the telecope of the observation
- settelescope - set the telecope of the observation
- observer - get observer name
- setobserver - set observer name

• **Utility -** There is a range of utility services available through the functions

  - axesmap - get mapping between pixel and world axes order
  - axiscoordinatetypes - get type of coordinate for each axis
  - coordinatetype - get type of coordinates
  - copy - make a copy of this tool
  - done - destroy this `tool`
  - findaxis - find specified axis (by number) in coordinate system
  - findcoordinate - find specified (by number) coordinate
  - fromrecord - set Coordinate System from a *Glish* record
  - id - get the fundamental identifier of this `tool`
  - naxes - get number of axes
  - ncoordinates - get the number of coordinates
  - reorder - reorder coordinates
  - summary - summarize the Coordinate System
  - torecord - Convert a Coordinate SYstem to a *Glish* record

- type - the type of this `tool`

- **Coordinate conversion**
  - convert - Convert one numeric coordinate with mixed input and output formats (abs/rel/world/pixel)
  - toabs - Convert a relative coordinate to an absolute coordinate
  - topixel - Convert from absolute world coordinate to absolute pixel coordinate
  - torel - Convert an absolute coordinate to a relative coordinate
  - toworld - Convert from an absolute pixel coordinate to an absolute world coordinate
  - convertmany - Convert many numeric coordinates with mixed input and output formats (abs/rel/world/pixel)
  - toabsmany - Convert many relative coordinates to absolute coordinates
  - topixelmany - Convert many absolute world coordinates to absolute pixel coordinates
  - torelmany - Convert many absolute coordinates to relative coordinates
  - toworldmany - Convert many absolute pixel coordinates to absolute world coordinates
  - frequencytovelocity - Convert from frequency to velocity
  - frequencytofrequency - Convert from frequency to frequency via a velocity offset
  - velocitytofrequency - Convert from velocity to frequency
  - setconversiontype - Set extra reference frame conversion layer
  - conversiontype - Recover extra reference frame conversion types

- **Tests -**
  - coordsystest - Run test suite for Coordsys `tool`

Events
There are no events emitted or acted upon by this `tool`.

Example
include 'coordsys.g'

```
cs := coordsys(direction=T, linear=2)
cs.summary()
```

coordsys - Constructor

Package	**general**
Module	**images**
Tool	**coordsys**

Create a default coordsys tool

Synopsis

coordsys(direction, spectral, stokes, linear, tabular)

Arguments

direction	in	Make a direction coordinate ? Allowed:Bool Default: F
spectral	in	Make a spectral coordinate ? Allowed:Bool Default: F
stokes	in	Make a Stokes coordinate with these Stokes Allowed:String or vector of Strings Default: ""
linear	in	Make a linear coordinate with this many axes Allowed:Integer Default: 0
tabular	in	Make a tabular coordinate Allowed:Bool Default: F

Returns

Coordsys `tool` or fail

Description

By default, this constructor makes an empty Coordsys `tool`. You can ask it to include various sorts of coordinates through the arguments. Except for Stokes, you don't have any control over the coordinate contents (e.g. reference value etc.) it does make for you on request. But you can edit the Coordinate System after creation if you wish.
If you wish to make a Stokes coordinate, then you assign `stokes` to a string (or a vector of strings) saying which Stokes you want. AIPS++ allows rather a lot of potential Stokes types.

Probably most useful is some combination of the basic I, Q, U, V, XX, YY, XY, YX, RR, LL, RL, and LR.
However, a more esoteric choice is also possible: RX, RY, LX, LY, XR, XL, YR, YL (these are mixed linear and circular), PP, PQ, QP, QQ (general quasi-orthogonal correlation products) RCircular, LCircular, Linear (single dish polarization types).
You can also specify some polarization 'Stokes' types: Ptotal (Polarized intensity $((Q^2+U^2+V^2)^(1/2))$, Plinear (Linearly Polarized intensity $((Q^2+U^2)^(1/2))$, PFtotal (Polarization Fraction (Ptotal/I)), PFlinear (Linear Polarization Fraction (Plinear/I)), and Pangle (Linear Polarization Angle $(0.5arctan(U/Q)$ in radians)).
Probably you will find the more unusual types aren't fully supported throughout the system.
You can make a LinearCoordinate with as many uncoupled axes as you like. Thus, `linear=2` makes one LinearCoordinate with 2 axes (think of it like a DirectionCoordinate which also has 2 axes [but coupled in this case], a longitude and a latitude).
If you make a TabularCoordinate, it is linear to start with. You can change it to a non-linear one by providing a list of pixel and world values to function settabular.

Example

```
- cs := coordsys()
- cs.ncoordinates()
0
- cs.done()
- cs
F
#
- cs2 := coordsys(direction=T, stokes='I V');
- cs2.ncoordinates()
2
- cs2.coordinatetype()
Direction Stokes
#
- cs2.summary()
Direction system : J2000
Telescope        : ATCA
Observer         : Karl Jansky
Date observation : 2000/07/03/08:10:57

Axis Type      Name             Proj   Coord value at pixel    Coord incr Units
--------------------------------------------------------------------------------
```

```
1    Direction Right Ascension   SIN  00:00:00.000     1.00 -6.000000e+01 arcsec
2    Direction Declination       SIN +00.00.00.000     1.00  6.000000e+01 arcsec
3    Stokes    Stokes                             IV
```

The second Coordinate System contains a direction coordinate and a Stokes coordinate. This means that there are three 'axes' associated with the 2 coordinates.

---

coordsys.axesmap - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Find mapping between world and pixel axes

Synopsis
axesmap(toworld)

Arguments

toworld	in	Map from pixel to world ? Allowed:T or F Default: T

Returns
Vector of Int or fail

Description

This function returns a vector describing the mapping from pixel to world or world to pixel axes. It is not for general user use.
See the discussion about pixel and world axis ordering. Generally they will be in the same order.

Example

```
- cs := coordsys(direction=T, spectral=T)
- cs.axesmap(T);
[1 2 3]
- cs.axesmap(F);
[1 2 3]
```

---

coordsys.axiscoordinatetypes - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Return types of coordinates for each axis

Synopsis
axiscoordinatetypes(world)

Arguments

world	in	World or pixel axes ? Allowed:T or F Default: T

Returns
Vector of strings or fail

Description

This function (short-hand name `act`) returns a vector string giving the coordinate type for each axis (world or pixel) in the Coordinate System. See the discussion about pixel and world axis ordering.

Example

```
- cs := coordsys(direction=T, spectral=T)
- cs.axiscoordinatetypes()
Direction Direction Spectral
```

---

coordsys.conversiontype - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Get extra reference conversion layer

Synopsis
conversiontype(type)

Arguments

type	in	Coordinate type Allowed:String from 'direction', 'spectral'

Returns
String or fail

Description

Some coordinates contain a reference code. Examples of reference codes are B1950 and J2000 for direction coordinates, or LSRK and BARY for spectral coordinates. When you do conversions between pixel and world coordinate, the coordinates are in the reference frame corresponding to these codes.
Function setconversiontype allows you to specify a different reference frame which is used when converting between world and pixel coordinate.
This function allows you to recover those conversion types. If no extra conversion layer has been set, you get back the native reference types.

Example

```
- cs := coordsys(direction=T, spectral=T)
- print cs.conversiontype (type='direction'), ' ', cs.conversiontype (type='spectral'),
J2000 LSRK
-
- cs.setconversiontype (direction='GALACTIC', spectral='BARY');
- print cs.conversiontype (type='direction'), ' ', cs.conversiontype (type='spectral'),
GALACTIC BARY
```

---

coordsys.convert - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert a numeric mixed coordinate

Synopsis

convert(coordin, absin, dopplerin, unitsin, absout, dopplerout, unitsout, shape)

Arguments

coordin	in	Input coordinate Allowed:Numeric vector
absin	in	Are input coordinate elements absolute ? Allowed:Boolean vector Default: All T
dopplerin	in	Input doppler type for velocities Allowed:String Default: 'radio'
unitsin	in	Input units Allowed:String vector Default: Native
absout	in	Are output coordinate elements absolute ? Allowed:Boolean vector Default: All T
dopplerout	in	Output doppler type for velocities Allowed:String Default: 'radio'
unitsout	in	Output units Allowed:String vector Default: Native
shape	in	Image shape Allowed:Integer vector Default: Unused

Returns

Numeric vector or fail

Description

This function converts between mixed pixel/world/abs/rel numeric coordinates. The input and output coordinates are specified via a numeric vector giving coordinate values, a string vector giving units, a boolean vector specifying whether the coordinate is absolute or relative (to the reference pixel) and doppler strings specifying the doppler convention for velocities.
The units string may include `pix` for pixel coordinates and velocity units (i.e. any unit consistent with `m/s`).
The allowed doppler strings and definition are described in function summary.
The `shape` argument is optional. If your Coordinate System from an image, then assign the image shape to this argument. It is used only when making mixed (pixel/world) conversions for Direction Coordinates to resolve ambiguity.
The example will make the use of this function clear.

Example

```
- cs := coordsys(direction=T, spectral=T)    # 3 axes
-
- cout := cs.convert (coordin=[10,20,30],
                      absin=[T,T,T],
                      unitsin="pix pix pix",
                      absout=[T,F,T],
                      dopplerout='optical',
                      unitsout="pix arcsec km/s");
-
- print 'cout=', cout
[10 1140.0058 1134.82336]
```

In this example we convert from a vector of absolute pixels to a mixture of pixel/world and abs/rel.

---

coordsys.convertmany - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert many numeric mixed coordinates

Synopsis
convertmany(coordin, absin, dopplerin, unitsin, absout, dopplerout, unitsout, shape)

Arguments

coordin	in	Input coordinate Allowed:Numeric matrix
absin	in	Are input coordinate elements absolute ? Allowed:Boolean vector Default: All T
dopplerin	in	Input doppler type for velocities Allowed:String Default: 'radio'
unitsin	in	Input units Allowed:String vector Default: Native
absout	in	Are output coordinate elements absolute ? Allowed:Boolean vector Default: All T
dopplerout	in	Output doppler type for velocities Allowed:String Default: 'radio'
unitsout	in	Output units Allowed:String vector Default: Native
shape	in	Image shape Allowed:Integer vector Default: Unused

Returns
Numeric matrix or fail

Description

This function converts between many mixed pixel/world/abs/rel numeric coordinates. See function convert for more information.
The only diffference with that function is that you provide a matrix holding many coordinates to convert and a matrix of many converted coordinates is returned.

Example

```
- cs := coordsys(direction=T, spectral=T)    # 3 axes
-
- p := [15,20,30];                           # absolute pixel coordinate
```

```
- cin := array (p, 3, 10);                  # 10 conversions each of length 3
- cin[3,] := 1:10;                          # spectral pixel runs from 1 to 10
- cin
[[1:3,]
    15 15 15 15 15 15 15 15 15 15
    20 20 20 20 20 20 20 20 20 20
    1  2  3  4  5  6  7  8  9  10]
-
- cout := cs.convertmany (coordin=cin,
                          absin=[T,T,T],
                          unitsin="pix pix pix",
                          absout=[T,F,T],
                          dopplerout='optical',
                          unitsout="pix deg km/s");
-
- cout
[[1:3,]
    15          15          15          15          15          15          15          15          15          15
    0.316668279 0.316668279 0.316668279 0.316668279 0.316668279 0.316668279 0.316668279 0.316668279 0.316668279 0.316668279
    1145.30291  1145.09023  1144.87756  1144.66488  1144.4522   1144.23953  1144.02685  1143.81418  1143.6015   1143.38883]
-
- cout[,1]                                    # First coordinate
[15 0.316668279 1145.30291]
-
- cout[,10]                                   # 10th coordinate
[15 0.316668279 1143.38883]
```

---

coordsys.coordinatetype - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Return type of specified coordinate

Synopsis
coordinatetype(which)

Arguments

which	in	Which coordinate ? Allowed:1-rel integer or unset Default: All

Returns
String or fail

Description

This function (short-hand name `ct`) returns a string describing the type of the specified coordinate. If `which=unset` the types for all coordinates are returned. Possible output values are 'Direction', 'Spectral', 'Stokes', 'Linear', and 'Tabular'

Example

```
- cs := coordsys(direction=T, spectral=T)
- cs.coordinatetype(1)
Direction
- cs.coordinatetype()
Direction Spectral
```

---

coordsys.copy - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Copy this Coordsys tool

Synopsis
copy()

Returns
Coordsys tool or fail

Description

Glish uses copy-on-write semantics. This sometimes means that it maintains a reference between variables that you would rather it didn't.
This function returns a copy, not a reference, of the Coordsys `tool`. It is your responsibility to call the `done` function on the new `tool`.

Example

```
- cs := coordsys()
- cs2 := cs            # Reference
- cs.done()
- cs
F
- cs2                  # cs2 gets doned when c2 does
F
-
- cs := coordsys()
- cs2 := cs.copy()     # Copy
- cs.done()
- cs
F
- cs2
ref [busy=<function>, result=<function>,...   # cs2 is still viable
- cs2.done()
- cs2                  # Now it's done
F
```

---

coordsys.done - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Destroy this Coordsys tool

Synopsis
done()

Returns
T or fail

Description

If you no longer need to use a Coordsys `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`.

Example

```
- cs := coordsys()
- cs.done()
- cs
F
```

---

coordsys.epoch - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Return the epoch

Synopsis
epoch()

Returns
Epoch measure or fail

Description

This function (short-hand name `e`) returns the epoch of the observation as a Measure. You can format it with quanta

Example

```
- cs := coordsys()
- ep := cs.epoch()
- ep
[type=epoch, refer=UTC, m0=[value=51728.4578, unit=d]]
-
- time := dm.getvalue(ep)           # Extract time with measures
- dq.time(time)                     # Format with quanta
10:59:10.170
```

---

coordsys.findaxis - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Find specified axis in coordinate system

Synopsis
findaxis(coordinate, axisincoordinate, world, axis)

Arguments

coordinate	out	Coordinate number Allowed:Integer
axisincoordinate	out	Axis in the coordinate Allowed:Integer
world	in	is axis a world or pixel axis ? Allowed:Bool Default: T
axis	in	Axis in coordinate system Allowed:Integer Default: 1

Returns
T, F or fail

Description

This function (short-hand name **fa**) finds the specified axis in the Coordinate System. If the axis does not exist, it returns F (not fail).

Example

```
- cs := coordsys(direction=T, linear=2)          # RA/DEC/Lin1/Lin2
- local coord, axisincoord;
- cs.findcoordinate(coord, axisincoord, T, 2)    # DEC
- print coord, axisincoord
1 2
- cs.findcoordinate(coord, axisincoord, T, 3)    # Lin1
- print coord, axisincoord
2 1
```

In these examples, the Coordinate System has 4 axes (both world and pixel the same). The first example finds the DEC axis (coordinate system axis 2) to be the second axis of the Direction Coordinate (coordinate 1). The second example finds the first linear axis (coordinate system axis 3) to be the first axis of the Linear Coordinate (coordinate 2).

---

coordsys.findcoordinate - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Find axes of specified coordinate

Synopsis

findcoordinate(pixel, world, type, which)

Arguments

pixel	out	Pixel axes Allowed:Vector of integers
world	out	World axes Allowed:Vector of integers
type	in	Type of coordinate to find Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" Default: "direction"
which	in	Which coordinate if more than one Allowed:Integer Default: First (1)

Returns

T, F or fail

Description

This function (short-hand name **fc**) finds the axes in the Coordinate System for the specified coordinate (minimum match is active for argument `type`). By default it finds the first coordinate, but if there is more than one (can happen for linear coordinates), you can specify which. If the coordinate does not exist, it returns F (not fail).

See also the function axesmap which returns the mapping between pixel and world axes.

Example

```
- cs := coordsys(direction=T)
- local pa, wa;
- cs.findcoordinate(pa, wa, 'direction')
- print 'pixel, world axes =', pa, wa
pixel, world axes = [1 2] [1 2]
```

---

coordsys.frequencytofrequency - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert frequency to frequency with an offset

Synopsis

frequencytofrequency(value, frequnit, velocity, doppler)

Arguments

value	in	Frequency to convert Allowed:Numeric or vector of numeric
frequnit	in	Unit of input frequencies Allowed:String Default: Intrinsic units
velocity	in	Velocity offset Allowed:Quantity Default: None
doppler	in	Velocity doppler definition Allowed:String Default: 'radio'

Returns

Double or vector of double or fail

Description

This function (short-hand name **ftf**) converts frequencies to frequencies by applying (subtracting) a velocity offset.
The input frequencies are specified via a vector of numeric values and a specified unit (`frequnit`). If you don't give a frequency unit, it is assumed that the units are those given by function units for the spectral coordinate.
This function does not make any frame conversions (e.g. LSR to BARY) but you can specifiy the velocity doppler definition for the velocity via the `doppler` argument (see summary for possible values).
This function will return a fail if there is no spectral coordinate in the Coordinate System. See also function frequencytovelocity.
This function presently uses an approximation only valid for velocities less than the speed of light. The correction tends to zero as the velocity appraches the speed of light. You will get NaNs for larger values of the velocity.

Example

```
- cs := coordsys(spectral=T)
- idx := 1 + 0.1*(1:5)
- rv := cs.referencevalue()
- freq := idx * rv
-
- print 'In = ', freq
[1.5565e+09 1.698e+09 1.8395e+09 1.981e+09 2.1225e+09]
-
- print cs.frequencytofrequency(freq, velocity='100km/s')
[1.55598089e+09 1.6974337e+09 1.83888651e+09 1.98033932e+09 2.12179213e+09]
```

---

coordsys.frequencytovelocity - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert frequency to velocity

Synopsis

frequencytovelocity(value, frequnit, doppler, velunit)

Arguments

value	in	Frequency to convert Allowed:Numeric or vector of numeric
frequnit	in	Unit of input frequencies Allowed:String Default: Intrinsic units
doppler	in	Velocity doppler definition Allowed:String Default: 'radio'
velunit	in	Unit of output velocities Allowed:String Default: 'km/s'

Returns
Double or vector of double or fail

Description

This function (short-hand name **ftv**) converts frequencies to velocities.
The input frequencies are specified via a vector of numeric values and a specified unit (`frequnit`). If you don't give a frequency unit, it is assumed that the units are those given by function units for the spectral coordinate.
This function does not make any frame conversions (e.g. LSR to BARY) but you can specifiy the velocity doppler definition via the `doppler` argument (see summary for possible values).
The velocities are returned in a vector for which you specify the units (`velunit` - default is km/s).
This function will return a fail if there is no spectral coordinate in the Coordinate System. See also function velocitytofrequency.

Example

```
- im := image('hcn.cube')
- cs := im.coordsys()
-
- local pa, wa;
- cs.findcoordinate(pa, wa, 'spectral')     # Find spectral axis
- pixel := cs.referencepixel();             # Use reference pixel for non-spectral
-
- nFreq := im.shape()[pa[1]];               # Length of spectral axis
- freq := [];
```

```
- for (i in 1:nFreq) {
    pixel[pa[1]] := i;                      # Assign value for spectral axis of pixel coordinate
    w := cs.toworld(value=pixel, format='n')# Convert pixel to world
    freq[i] := w[wa[1]];                    # Fish out frequency
  }
- vel := cs.frequencytovelocity(value=freq, doppler='optical', velunit='km/s')
- print vel
[137.868718 133.690916 129.513229 125.33566 121.158206 116.980869
 112.803649 108.626544 104.449556 100.272685 96.0959294 91.9192905
  87.7427679 83.5663616 79.3900717 75.213898 71.0378408 66.8618999
  62.6860752 58.5103668 54.3347748 50.159299 45.9839395 41.8086963
  37.6335694 33.4585588 29.2836644 25.1088863 20.9342244 16.7596788
  12.5852495  8.41093638 4.23673952 0.0626588884 -4.11130551 -8.28515368
 -12.4588856 -16.6325014 -20.8060009 -24.9793842 -29.1526513 -33.3258023
 -37.498837 -41.6717555 -45.8445579 -50.0172441 -54.1898141 -58.3622679
 -62.5346056 -66.7068272 -70.8789325 -75.0509218 -79.2227948 -83.3945518
 -87.5661926 -91.7377173 -95.9091259 -100.080418 -104.251595 -108.422655
-112.593599 -116.764427 -120.935139 -125.105735]
```

In this example, we find the optical velocity in km/s of every pixel along the spectral axis of our image. First we obtain the Coordinate System from the image. Then we find which axis of the Coordinate System (image) pertain to the spectral coordinate. Then we loop over each pixel of the spectral axis, and convert a pixel coordinate (one for each axis of the image) to world. We obtain the value for the spectral axis from that world vector, and add it to the vector of frequencies. Then we convert that vector of frequencies to velocity.

---

coordsys.fromrecord - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Fill Coordinate System from a record

Synopsis

fromrecord(record)

Arguments

record	in	Record containing Coordinate System Allowed:Glish record

Returns
T or fail

Description

You can convert a Coordinate System to a record (torecord). This function (fromrecord) allows you to set the contents of an existing Coordinate System from such a record. In doing so, you overwrite its current contents.
These functions are not for general user use.

Example

```
- cs := coordsys(direction=T, stokes="I Q")
- cs.ncoordinates()
2
- r := cs.torecord()
-
- cs2 := coordsys()
- cs2.ncoordinates()
0
- cs2.fromrecord(r)
- cs2.ncoordinates()
2
```

---

coordsys.id - Function

Package	**general**
Module	**images**
Tool	**coordsys**

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` used in the tasking system.

Returns
ToolID - a record

Example

```
- cs := coordsys()
- cs.id()
[sequence=5, pid=17277, time=962622872, host=phoenix, agentid=3]
```

---

coordsys.increment - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the increments

Synopsis
increment(format, type)

Arguments

format	in	Format string Allowed:String from combination of "n", "q", "s", "m" Default: 'n'
type	in	Coordinate type Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All

Returns
Vector of doubles, vector of quantities, vector of string, record or fail

Description

Each axis associated with the Coordinate System has a reference value, reference pixel and an increment (per pixel). These are used in the mapping from pixel to world coordinate.

This function (short-hand name i) returns the increment (in world axis order). You can recover the increments either for all coordinates (leave `type` unset) or for a specific coordinate type (mimumum match of the allowed types will do). If you ask for a non-existent coordinate an exception is generated.
See the discussion regarding the formatting possibilities available via argument `format`.
You can set the increment with function setincrement.

Example

```
- cs := coordsys(direction=T, spectral=T)
- cs.increment(format='q')
[__*0=[value=-1, unit='], __*1=[value=1, unit='], __*2=[value=1000, unit=Hz]]
-
- cs.increment(format='n')
[-1 1 1000]
-
- cs.increment(format='n', type='spectral')
1000
```

---

coordsys.lineartransform - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the linear transform matrix

Synopsis
lineartransform(type)

Arguments

type	in	Coordinate type Allowed:String from "direction", "stokes", "spectral", "linear", "tabular"

Returns
Vector of double or fail

Description

Recover the linear transform component for the specified coordinate type. You can set the linear transform with function setlineartransform.

Example

```
- cs := coordsys(direction=T, linear=3)
- cs.lineartransform('dir')                # 2 x 2
[[1:2,]
    1 0
    0 1]
- cs.lineartransform('lin')                # 3 x 3
[[1:3,]
    1 0 0
    0 1 0
    0 0 1]
```

---

coordsys.names - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the names for each axis

Synopsis
names(type)

Arguments

type	in	Coordinate type Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All

Returns
Vector of strings or fail

Description

Each axis associated with the Coordinate System has a name (they don't mean anything fundamental). This function returns those names in world axis order. You can recover the names either for all coordinates (leave `type` unset) or for a specific coordinate type (mimumum match of the allowed types will do). If you ask for a non-existent coordinate an exception is generated.
You can set the names with function setnames.

Example

```
- cs := coordsys(direction=T, spectral=T)
- n := cs.names(T)
- n[1]
Right Ascension
- n[2]
Declination
- n[3]
Frequency
-
- cs.names('spec')
Frequency
```

---

coordsys.naxes - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the number of axes

Synopsis
naxes(world)

Arguments

world	in	Find number of world or pixel axes ? Allowed:T or F Default: T

Returns
Int or fail

Description

Find the number of axes in the Coordinate System.
You may find the number of world or pixel axes; these are generally the same and general users can ignore the distinction. See the discussion about pixel and world axis ordering.

Example

```
- cs := coordsys(direction=T, spectral=T)
- n := cs.naxes(T)
3                          # 2 direction axes, 1 spectral
- n := cs.naxes(F)
3
```

---

coordsys.ncoordinates - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the number of coordinates in the Coordinate System

Synopsis
ncoordinates()

Returns
Integer or fail

Description

This function (short-hand name `nc`) recovers the number of coordinates in the Coordinate System.

Example

```
- cs := coordsys(direction=T, spectral=T)
- cs.ncoordinates()
2
- cs2 := coordsys(linear=4)
- cs2.ncoordinates()
1
```

---

coordsys.observer - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Return the name of the observer

Synopsis
observer()

Returns
String or fail

Description

This function returns the name of the observer. You can set it with the function setobserver.

Example

```
- cs := coordsys()
- cs.observer()
Karl Jansky
```

---

coordsys.projection - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the direction coordinate projection

Synopsis
projection(type)

Arguments

type	in	Type of projection Allowed:String Default: Current

Returns
Record, Vector string, Int or fail

Description

If the Coordinate System contains a direction coordinate, this function (short-hand name `p`) can be used to recover information about the projection. For discussion about celestial coordinate systems, including projections, see the papers by Mark Calabretta and Eric Greisen. The initial draft from 1996 (implemented in AIPS++Background information can be found here.
What this function returns depends upon the value you assign to `type`.

- `type=unset`. In this case (the default), the actual projection type and projection parameters are returned in a *Glish* record with fields `type` and `parameters`, respectively.
- `type='all'`. In this case, a vector of strings containing all of the possible projection codes is returned.
- `type=code`. If you specify a valid projection type code (see list by setting `type='all'`) then what is returned is the number of parameters required to describe that projection (useful in function setprojection).

You can change the projection with setprojection.
If the Coordinate System does not contain a direction coordinate, an exception is generated.

Example

```
- cs := coordsys(direction=T)
- cs.projection()
[type=SIN, parameters=[0 0] ]
-
- cs.projection('all')
AZP TAN SIN STG ARC ZPN ZEA AIR CYP CAR MER CEA COP
COD COE COO BON PCO GLS PAR AIT MOL CSC QSC TSC
-
- cs.projection('ZPN')
10
```

We first recover the projection type and parameters from the direction coordinate. Then we find the list of all possible projection types. FInally, we recover the number of parameters required to describe the 'ZPN' projection.

---

coordsys.referencecode - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Return specified reference code

Synopsis
referencecode(type, list)

Arguments

type	in	Which coordinate type ? Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All
list	in	List all possibilities ? Allowed:Bool Default: F

Returns
Vector of strings, string or fail

Description

This function (short-hand name `rp`) returns the reference code for all, or the specified coordinate type. Examples of the reference code are B1950 and J2000 for direction coordinates, or LSRK and BARY for spectral coordinates.
If `type` is left unset, then a vector of strings is returned, one code for each coordinate type in the Coordinate System.
If you specify `type` then select from 'direction', 'spectral', 'stokes', and 'linear' (the first two letters will do). However, only the first two coordinate types will return a non-empty string. If the Coordinate System does not contain a coordinate of the type you specify, an exception is generated.
The argument `list` is ignored unless you specify a specific `type`. If `list=T`, then this function returns the list of all possible reference codes for the specified coordinate type. Otherwise, it just returns the actual code current set in the Coordinate System.
The list of all possible types is returned as a *Glish* record (it is actually generated by the listcodes function in the measures system). This record has two fields. These are called 'normal' (containing all normal codes) and 'extra' (maybe empty, with all extra codes like planets).
You can set the reference code with setreferencecode.

Example

```
- cs := coordsys(direction=T)
- list := cs.referencecode('dir', T)
- list.normal[1:10]
J2000 JMEAN JTRUE APP B1950 BMEAN BTRUE GALACTIC HADEC AZEL
-
- cs.referencecode('dir')
J2000
```

In this example we first get the list of all possible reference codes for a direction coordinate (and print some of the normal ones). Then we get the actual reference code for the direction coordinate in our Coordinate System.

---

coordsys.referencepixel - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the reference pixel

Synopsis

referencepixel(type)

Arguments

type	in	Coordinate type Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All

Returns

Vector of double or fail

Description

Each axis associated with the Coordinate System has a reference value, reference pixel and an increment (per pixel). These are used in the mapping from pixel to world coordinate.
This function (short-hand name `rp`) returns the reference pixel (in pixel axis order). You can recover the reference pixel either for all coordinates (leave `type` unset) or for a specific coordinate type (mimumum match of the allowed types will do). If you ask for a non-existent coordinate an exception is generated.
You can set the reference pixel with function setreferencepixel.

Example

```
- cs := coordsys(spectral=T, linear=2)
- cs.setreferencepixel([1.0, 2.0, 3.0])
- cs.referencepixel()
[1 2 3]
-
- cs.referencepixel('lin')
[2 3]
```

---

coordsys.referencevalue - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the reference value

Synopsis
referencevalue(format, type)

Arguments

format	in	Format string Allowed:String from combination of "n", "q", "s", "m" Default: 'n'
type	in	Coordinate type Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All

Returns
Vector of doubles, vector of quantities, vector of string, record or fail

Description

Each axis associated with the Coordinate System has a reference value, reference pixel and an increment (per pixel). These are used in the mapping from pixel to world coordinate.
This function (short-hand name `rv`) returns the reference value (in world axis order). You can recover the reference value either for all coordinates (leave `type` unset) or for a specific coordinate type (mimumum match of the allowed types will do). If you ask for a non-existent coordinate an exception is generated.
See the discussion regarding the formatting possibilities available via argument `format`.
You can set the reference value with function setreferencevalue.

Example

```
- cs := coordsys(direction=T, spectral=T)
- cs.referencevalue(format='q')
[__*0=[value=0, unit='], __*1=[value=0, unit='], __*2=[value=1.415e+09, unit=Hz]]
-
- cs.referencevalue(format='n')
[0 0 1.415e+09]
-
```

```
- cs.referencevalue(format='n', type='spec')
1.415e+09
```

---

coordsys.reorder - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Reorder the coordinates

Synopsis
reorder(order)

Arguments

order	in	New coordinate order Allowed:Vector of integers

Returns
Vector of quantities or doubles, or fail

Description

This function reorders the coordinates in the Coordinate System. You specify the new order of the coordinates in terms of their old order.

Example

```
- cs := coordsys(direction=T, spectral=T, linear=2)
- cs.coordinatetype()
Direction Spectral Linear
-
- cs.reorder([2,3,1]);
- cs.coordinatetype()
Spectral Linear Direction
```

---

coordsys.replace - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Replace a coordinate

Synopsis
replace(csys, whichin, whichout)

Arguments

csys	in	Coordinate System to replace from Allowed:Coordsys tool
whichin	in	Index of input coordinate Allowed:Integer
whichout	in	Index of output coordinate Allowed:Integer

Returns
T or fail

Description

This function replaces on coordinate in the current Coordinate System by one coordinate in the given Coordinate System. The specified coordinates must have the same number of axes.

Example

```
- cs := coordsys(direction=T, linear=1)
- cs.coordinatetype()
Direction Linear
- cs2 := coordsys(spectral=T)
- cs2.replace (cs2,2,1)
- cs.coordinatetype()
Direction Spectral
```

---

coordsys.restfrequency - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the rest frequency

Synopsis
restfrequency()

Returns
Quantity or fail

Description

If the Coordinate System contains a spectral coordinate, then it has a rest frequency. In fact, the spectral coordinate can hold several rest frequencies (to handle for example, an observation where the band covers many lines), although only one is active (for velocity conversions) at a time.
This function (short-hand name `rf`) recovers the rest frequencies as a quantity vector. The first frequency is the active one.
You can change the rest frequencies with setrestfrequency.
If the Coordinate System does not contain a frequency coordinate, an exception is generated.

Example

```
- cs := coordsys(spectral=T)
- cs.restfrequency()
[value=1.42040575e+09, unit=Hz]
-
- cs.setrestfrequency (value=[1.2e9. 1.3e9], which=2, append=F)
- cs.restfrequency()
[value=[1.3e+09 1.2e+09] , unit=Hz]
```

In the example, the initial spectral coordinate has 1 rest frequency. Then we set it with two, nominating the second as the active rest frequency, and recover them.

---

coordsys.setconversiontype - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set extra reference conversion layer

Synopsis
setconversiontype(direction, spectral)

Arguments

direction	in	Reference code Allowed:String or unset Default: None
spectral	in	Reference code Allowed:String or unset Default: None

Returns
T, F or fail

Description

Some coordinates contain a reference code. Examples of reference codes are B1950 and J2000 for direction coordinates, or LSRK and BARY for spectral coordinates. When you do conversions between pixel and world coordinate, the coordinates are in the reference frame corresponding to these codes.
This function allows you to specify a different reference frame which is used when converting between world and pixel coordinate (see function conversiontype to recover the conversion types). If it returns F, it means that although the conversion machines were successfully created, a trial conversion failed. This usually means the REST frame was involved which requires a radial velocity (not yet implemented). If this happens, the conversion type will left as it was. You will get a fail if more blatant things are wrong like a missing coordinate, or an incorrect reference code.
The list of possible reference codes can be obtained via function referencecode.
With this function, you specify the desired reference code. Then, when a conversion between pixel and world is requested, an extra conversion is done to (`toWorld`) or from (`toPixel`) the specified reference frame.

The summary function shows the extra conversion reference system to the right of the native reference system (if it is different) and in parentheses. Note that to convert between different spectral reference frames, you need a position, epoch and direction. The position (telescope) and epoch (date of observation), if not in your coordinate system can be set with functions settelescope and setepoch. The direction is the reference direction of the *required* direction coordinate in the coordinate system.
As an example, let us say you are working with a spectral coordinate which was constructed with the LSRK reference frame. You want to convert some pixel coordinates to barycentric velocities (reference code BARY).

```
- cs := coordsys(direction=T, spectral=T); # Create coordinate system
- local pa, wa;
- cs.findcoordinate(pa, wa, 'spectral')    # Find spectral coordinate
- u := cs.units()[wa[1]];                  # Spectral unit
- cs.referencecode(type='spectral')        # Which is  in LSRK reference frame
LSRK
-
- p := [10,20,30]
- w := cs.toworld(p, format='n')           # Convert a pixel to LSRK world
- print 'pixel, world = ', p, w
pixel, world =  [10 20 30]  [21590.9999 19.0000967 1.415029e+09]
-
- p2 := cs.topixel(w)                      # and back to pixel
- print 'world, pixel = ', w, p2
world, pixel =  [21590.9999 19.0000967 1.415029e+09]  [10 20 30]
-
- v := cs.frequencytovelocity(value=w[wa[1]],     # Convert LSRK frequency to LSRK velocity
                              frequnit=u,
                              doppler='RADIO',
                              velunit='m/s');
- print 'pixel, frequency, velocity = ', p[pa[1]], w[wa[1]], v
pixel, frequency, velocity =  30 1.415029e+09 1134823.36
-
-
- cs.setconversiontype(spectral='BARY')    # Specify BARY reference code
- w := cs.toworld(p, format='n')           # Convert a pixel to BARY world
print 'pixel, world = ', p, w
pixel, world =  [10 20 30]  [21590.9999 19.0000967 1.41498179e+09]
-
- p2 := cs.topixel(w)                      # and back to pixel
- print 'world, pixel = ', w, p2
world, pixel =  [21590.9999 19.0000967 1.41498179e+09]  [10 20 29.9952772]
-
- v := cs.frequencytovelocity(value=w[wa[1]],   # Convert BARY frequency to BARY velocity
                              frequnit=u,
```

```
                              doppler='RADIO',
                              velunit='m/s');
- print 'pixel, frequency, velocity = ', p[pa[1]], w[wa[1]], v
pixel, frequency, velocity =  30 1.41498179e+09 1144787.26
```

You must also be aware of when this extra layer is active and when it is not. It's a bit nasty.

- - Whenever you use `toWorld`, `toPixel` `toWorldMany`, or `toPixelMany` the layer is active.
- - Whenever you use `convert` or `convertMany` the layer *may* be active. Here are the rules !

  It is only relevant to spectral and direction coordinates.

  For the direction coordinate part of your conversion, if you request a pure world or pixel conversion it is active. Any pixel/world mix will not invoke it (because it is ill defined).

  For the spectral coordinate part it is always active (only one axis so must be pixel or world).
- - This layer is irrelevant to all functions converting between frequency and velocity, and absolute and relative. The values are in whatever frame you are working with.

The

summary function lists the reference frame for direction and spectral coordinates. If you have also set a conversion reference code it also lists that (to the right in parentheses).

---

coordsys.setdirection - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set direction coordinate values

Synopsis
setdirection(refcode, proj, projpar, refpix, refval, incr, xform, poles)

Arguments

refcode	in	Reference code Allowed: String Default: No change
proj	in	Projection type Allowed: String Default: No change
projpar	in	Projection parameters Allowed: Numeric vector Default: No change
refpix	in	Reference pixel Allowed: Numeric vector Default: No change
refval	in	Reference value Allowed: Vector of doubles, vector of quantities, quantity of vector of doubles, vector of string, record Default: No change
incr	in	Increment Allowed: Vector of doubles, vector of quantities, quantity of vector of doubles, vector of string, record Default: No change
xform	in	Linear transform Allowed: 2-D numeric array Default: No change
poles	in	Native poles Allowed: Vector of doubles, vector of quantities, quantity of vector of doubles, vector of string, record Default: No change

Returns

T or fail

Description

When you construct a Coordsys `tool`, if you include a Direction Coordinate, it will have some default parameters. This function simply allows you to replace the values of the Direction Coordinate.
You can also change almost all of those parameters (such as projection, reference value etc.) via the individual functions setreferencecode, setprojection, setreferencepixel, setreferencevalue, setincrement, and setlineartransform provided by the Coordsys `tool`. See those functions for

more details about the formatting of the above function arguments.
Bear in mind, that if your Coordinate System came from a real image, then the reference pixel is special and you should not change it.

Example

```
- cs := coordsys(direction=T);
- cs.setdirection (refcode='GALACTIC', proj='SIN', projpar=[0,0],
                   refpix=[-10,20], refval="10deg -20deg");
-
- cs.projection()
[type=SIN, parameters=[0 0] ]
- cs.referencepixel()
[-10 20]
- cs.referencevalue(format='s')
10.000000 deg -20.000000 deg
```

---

coordsys.setepoch - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set a new epoch

Synopsis
setepoch(value)

Arguments

value	in	New epoch Allowed:Epoch measure

Returns
T or fail

Description

This function (short-hand name `se`) sets a new epoch (supplied as an epoch measure) of the observation. You can get the current epoch with function epoch.

Example

```
- cs := coordsys()
- ep := cs.epoch()
- ep
[type=epoch, refer=UTC, m0=[value=51728.4578, unit=d]]
-
- ep := dm.epoch('UTC', 'today')
- cs.setepoch(ep)
- cs.epoch()
[type=epoch, refer=UTC, m0=[value=51729.1679, unit=d]]
```

---

coordsys.setincrement - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the increment

Synopsis
setincrement(value, type)

Arguments

value	in	Increments Allowed: Vector of doubles, vector of quantities, quantity of vector of doubles, vector of string, record
type	in	Which coordinate type ? Allowed: String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All

Returns
T or fail

Description

Each axis associated with the Coordinate System has a reference value, reference pixel and an increment (per pixel). These are used in the mapping from pixel to world coordinate.
This function (short-hand name `si`) allows you to set a new increment. You should not do this on "stokes" axes unless you are an adept or a big risk taker.
You can set the increments either for all axes (`type=unset`) or for just the axes associated with a particular coordinate type.
You may supply the increments in all of the formats described in the formatting discussion.
In addition, you can also supply the increments as a quantity of vector of doubles. For example `dq.quantity([-1,2],'arcsec')`.
You can recover the current increments with function increment.

Example

```
- cs := coordsys(direction=T, spectral=T)
- rv := cs.increment(format='q')
- rv
[__*0=[value=-1, unit='], __*1=[value=1, unit='], __*2=[value=1000, unit=Hz]]
-
- rv2 := dq.quantity('4kHz');
- cs.setincrement(value=rv2, type='spec')
- cs.increment(type='spec', format='q')
[value=4000, unit=Hz]
-
- cs.increment(format='q')
[__*0=[value=-1, unit='], __*1=[value=1, unit='], __*2=[value=4000, unit=Hz]]
-
- cs.setincrement (value="-2' 2' 2e4Hz")
- cs.increment(format='q')
[__*0=[value=-2, unit='], __*1=[value=2, unit='], __*2=[value=20000, unit=Hz]]
```

In the example we first recover the increments as a vector of quantities. We then create a quantity for a new value for the spectral coordinate increment. Note we use units of kHz whereas the spectral coordinate is currently expressed in units of Hz. We then set the increment for the spectral coordinate. We then recover the increment again; you can see 4kHz has been converted to 4000Hz. We also show how to set the increment using a string interface.

---

coordsys.setlineartransform - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the linear transform

Synopsis
setlineartransform(type, value)

Arguments

type	in	Which coordinate type ? Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" Default: All
value	in	Linear transform Allowed:Array of double

Returns
T or fail

Description

This function set the linear transform component. For Stokes Coordinates this function will return T but do nothing.
You can recover the current linear transform with function lineartransform.

Example

```
- cs := coordsys(spectral=T, linear=3)
- xf := cs.lineartransform('lin')
- xf
[[1:3,]
    1 0 0
    0 1 0
    0 0 1]
- xf[1,2] := 0.01
- cs.slt(xf,'lin')
- cs.lt('lin')
```

```
[[1:3,]
    1 0.01 0
    0 1    0
    0 0    1]
```

---

coordsys.setnames - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the axis names

Synopsis
setnames(value, type)

Arguments

value	in	Names Allowed:Vector of strings
type	in	Which coordinate type ? Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All

Returns
T or fail

Description

Each axis associated with the Coordinate System has a name. It isn't used in any fundamental way.
This function (short-hand name `sn`) allows you to set new axis names.
You can set the names either for all axes (`type=unset`) or for just the axes associated with a particular coordinate type.
You can recover the current axis names with function names.

Example

```
- cs := coordsys(spectral=T, linear=2)
- cs.setnames(value="a b c")
- cs.names()
a b c
-
- cs.setnames('lin', "flying fish")
- cs.names()
a flying fish
```

---

coordsys.setobserver - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set a new observer

Synopsis
setobserver(value)

Arguments

value	in	New observer Allowed:String

Returns
T or fail

Description

If you want grab all the glory, or transfer the blame, this function (short-hand name `so`) sets a new observer of the observation. You can get the current observer with function observer. The observer's name is not fundamental to the Coordinate System !

Example

```
- cs := coordsys()
- cs.observer()
Karl Jansky
- cs.setobserver('Ronald Biggs')
- cs.observer()
Ronald Biggs
```

---

coordsys.setprojection - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the direction coordinate projection

Synopsis
setprojection(type, parameters)

Arguments

type	in	Type of projection Allowed:String
parameters	in	Projection parameters Allowed:Vector of double Default: []

Returns
T or fail

Description

If the Coordinate System contains a direction coordinate, this function (short-hand name `sp`) can be used to set the projection. For discussion about celestial coordinate systems, including projections, see the papers by Mark Calabretta and Eric Greisen. The initial draft from 1996 (implemented in AIPS++) can be found here.
You can use the function projection to find out all the possible types of projection. You can also use it to find out how many parameters you need to describe a particular projection. See Calabretta and Greisen for details about

those parameters (see section 4 of their paper); in FITS terms these parameters are what are labelled as PROJP.
Some brief help here on the more common projections in astronomy.

- SIN has either 0 parameters or 2. For coplanar arrays like East-West arrays, one can use what is widely termed the NCP projection. This is actually a SIN projection where the parameters are 0 and $1/tan(\delta_0)$ where δ_0 is the reference declination. Images made from the ATNF's Compact Array with AIPS++ will have such a projection. Otherwise, the SIN projection requires no parameters (but you can give it two each of which is zero if you wish).
- TAN is used widely in optical astronomy. It requires 0 parameters.
- ZEA (zenithal equal area) is used widely in survey work. It requires 0 parameters.

If the Coordinate System does not contain a direction coordinate, an exception is generated.

Example

```
- im := image('cena')
- cs := im.coordsys()
- cs.projection()
[type=SIN, parameters=[0 0] ]
-
- cs.projection('ZEA')
0
- cs.setprojection('ZEA')
- im2 := im.regrid('cena.zea', csys=cs)
```

We change the projection of an image from SIN to ZEA (which requires no parameters).

---

coordsys.setreferencecode - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set new reference code

Synopsis
setreferencecode(value, type, adjust)

Arguments

value	in	Reference code Allowed:String
type	in	Which coordinate type ? Allowed:String from "direction" or "spectral"
adjust	in	Adjust reference value ? Allowed:Bool Default: T

Returns
T or fail

Description

This function (short-hand name `src`) sets the reference code for the specified coordinate type. Examples of reference codes are B1950 and J2000 for direction coordinates, or LSRK and BARY for spectral coordinates.
You must specify `type`, selecting from 'direction', or 'spectral' (the first two letters will do). If the Coordinate System does not contain a coordinate of the type you specify, an exception is generated.
Specify the new code with argument `value`. To see the list of possible codes, use the function referencecode (see example).
If `adjust` is T, then the reference value is recomputed. This is invariably the correct thing to do. If `adjust` is F, then the reference code is simply overwritten; do this very carefully.

Example

```
- cs := coordsys(direction=T)
- list := cs.referencecode('dir', T)      # See possibilities
- list.normal[1:10]
J2000 JMEAN JTRUE APP B1950 BMEAN BTRUE GALACTIC HADEC AZEL
-
- cs.referencecode('dir')
J2000
- cs.setreferencecode('B1950', 'dir', T)
```

In this example we first get the list of all possible reference codes for a direction coordinate (and print some of the normal ones). Then we set the actual reference code for the direction coordinate in our Coordinate System.

Example

```
- cs := coordsys(direction=T)
- list := cs.referencecode('dir', T)      # See possibilities
- list.normal[1:10]
J2000 JMEAN JTRUE APP B1950 BMEAN BTRUE GALACTIC HADEC AZEL
-
- cs.referencecode('dir')
J2000
- cs.setreferencecode('B1950', 'dir', T)
```

In this example we first get the list of all possible reference codes for a direction coordinate (and print some of the normal ones). Then we set the actual reference code for the direction coordinate in our Coordinate System.

Example

```
- im := image('myimage.j2000')                        # Open image
- cs := im.coordsys()              # Get Coordinate System
- cs.referencecode('dir', F)
J2000
- cs.setreferencecode('B1950', 'dir', T)              # Set new direction system
- im2 := im.regrid(outfile='myimage.b1950', csys=cs)  # Regrid and make new image
```

In this example we show how to regrid an image from J2000 to B1950. First we recover the Coordinate System into the Coordsys `tool` called **cs**. We then set a new direction reference code, making sure we recompute the reference value. Then the new Coordinate System is supplied in the regridding process (done with an Image `tool`).

---

coordsys.setreferencelocation - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set reference pixel and value

Synopsis
setreferencelocation(pixel, world, mask)

Arguments

pixel	in	New reference pixel Allowed: Vector of integer Default: Old reference pixel
world	in	New reference value Allowed: Vector of doubles, vector of quantities, quantity of vector of doubles, vector of string, record Default: Old reference value
mask	in	Indicates which axes to center Allowed: Vector of boolean Default: All

Returns
T or fail

Description

This function (short-hand name `srl`) sets the reference pixel and reference value to the specified values. The world coordinate can be specified in any of the formats that the output world coordinate is returned in by the toworld function.
You can specify a mask (argument `mask`) indicating which pixel axes are set (T) and which are left unchanged (F). This function will refuse to change the reference location of a Stokes axis (gets you into trouble otherwise).
This function can be rather useful when regridding images. It allows you to keep easily a particular feature centered in the regridded image.

Example

```
- cs := coordsys(linear=2)
- cs.referencepixel()
[1 1]
- cs.referencevalue()
```

```
[0 0]
-
- w := cs.toworld([20,20], format='n')
- shp := [129,129]
- p := (shp-1)/2.0 + 1;
- cs.setreferencelocation (pixel=p, world=w)
- cs.referencepixel()
[65 65]
- cs.referencevalue()
[19 19]
```

---

coordsys.setreferencepixel - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the reference pixel

Synopsis
setreferencepixel(type, value)

Arguments

type	in	Which coordinate type ? Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All
value	in	Reference pixel Allowed:Vector of double

Returns
T or fail

Description

Each axis associated with the Coordinate System has a reference value, reference pixel and an increment (per pixel). These are used in the mapping from pixel to world coordinate.

This function (short-hand name `srp`) allows you to set a new reference pixel. You should not do this on "stokes" axes unless you are an adept or a big risk taker.
You can set the reference pixel either for all axes (`type=unset`) or for just the axes associated with a particular coordinate type.
Bear in mind, that if your Coordinate System came from a real image, then the reference pixel is special and you should not change it for Direction Coordinates.
You can recover the current reference pixel with function referencepixel.

Example

```
- cs := coordsys(spectral=T, linear=2)
- cs.setreferencepixel(value=[1.0, 2.0, 3.0])
- cs.referencepixel()
[1 2 3]
-
- cs.setreferencepixel('lin', [-1, -1])
- cs.referencepixel()
[1 -1 -1]
```

---

coordsys.setreferencevalue - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the reference value

Synopsis
setreferencevalue(value, type)

Arguments

value	in	Reference value Allowed: Vector of doubles, vector of quantities, quantity of vector of doubles, vector of string, record
type	in	Which coordinate type ? Allowed: String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All

Returns
T or fail

Description

Each axis associated with the Coordinate System has a reference value, reference pixel and an increment (per pixel). These are used in the mapping from pixel to world coordinate.
This function (short-hand name `srv`) allows you to set a new reference value.
You should not do this on "stokes" axes unless you are an adept or a big risk taker.
You may supply the reference value in all of the formats described in the formatting discussion.
You can recover the current reference value with function referencevalue.
Note that the value argument should be one of the specified possibilitioes.
Especially a `measure` will be accepted, but will have a null effect, due to the interpretation as a generic record.

Example

```
- cs := coordsys(direction=T, spectral=T)
- rv := cs.referencevalue(format='q')
- rv
[__*0=[value=0, unit='], __*1=[value=0, unit='], __*2=[value=1.415e+09, unit=Hz]]
-
- rv2 := rv[3];
- rv2.value := 2.0e9;
- cs.setreferencevalue(type='spec', value=rv2)
- cs.referencevalue(format='n')
[0 0 2e+09]
```

To set a new direction reference value, the easiest way, given a direction measure `dr` would be:

```
- cs.setreferencevalue(type='direction', value=dm.getvalue(dr))
```

In the example we first recover the reference value as a vector of quantities. We then select out the value for the spectral coordinate, change it, and set it back in the Coordinate System. We then recover the reference value again, but this time just as a vector of doubles. In the case of the dircetion, we set the value from a known direction measure.

---

coordsys.setrestfrequency - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the rest frequency

Synopsis
setrestfrequency(value, which, append)

Arguments

value	in	New rest frequencies Allowed:Quantity, double or string vector
which	in	Which is the active rest frequency Allowed:Integer Default: 1
append	in	Append this list or overwrite ? Allowed:Bool Default: F

Returns
T or fail

Description

If the Coordinate System contains a spectral coordinate, then it has a rest frequency. In fact, the spectral coordinate can hold several rest frequencies (to handle for example, an observation where the band covers many lines), although only one is active (for velocity conversions) at a time.
This function (short-hand name `srf`) allows you to set new rest frequencies. You can provide the rest frequency as a quantity, a quantity string, or a double (units of current rest frequency assumed).

You specify whether the list of frequencies will be appended to the current list or whether it will replace that list. You must select which of the frequencies will become the active one. By default its the first in the list. The index refers to the final list (either appended or replaced).
You can recover the current rest frequencies with restfrequency.
If the Coordinate System does not contain a frequency coordinate, an exception is generated.

Example

```
- cs := coordsys(spectral=T)
- cs.restfrequency()
[value=1.42040575e+09, unit=Hz]
-
- cs.setrestfrequency(dq.quantity('1.4GHz'))
- cs.restfrequency()
[value=1.4e+09, unit=Hz]
-
- cs.setrestfrequency(1.3e9)
- cs.restfrequency()
[value=1.3e+09, unit=Hz]
-
- cs.setrestfrequency (value=[1.2e9, 1.3e9], which=2)
- cs.restfrequency()
[value=[1.3e+09 1.2e+09] , unit=Hz]
-
- cs.setrestfrequency (dq.quantity([1,2],'GHz'), which=4, append=T)
- cs.restfrequency()
[value=[2e+09 1.2e+09 1.3e+09 1e+09] , unit=Hz]
-
- cs.setrestfrequency ("1.4E9Hz 1667MHz")
- cs.restfrequency()
[value=[1.4e+09 1.667e+09] , unit=Hz]
```

---

coordsys.setspectral - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set tabular values for the spectral coordinate

Synopsis
setspectral(refcode, restfreq, frequencies, doppler, velocities)

Arguments

refcode	in	Reference code Allowed:String Default: No change
restfreq	in	Rest frequency Allowed:Quantum or double Default: No change
frequencies	in	Vector of frequencies Allowed:Quantum vector Default: No change
doppler	in	Doppler type Allowed:String Default: No change
velocities	in	Vector of velocitiestypes Allowed:Quantum vector Default: No change

Returns
T or fail

Description

When you construct a Coordsys `tool`, if you include a Spectral Coordinate, it will be linear in frequency. This function allows you to replace the Spectral Coordinate by a finite table of values. Coordinate conversions between pixel and world are then done by interpolation.

You may specify either a vector of frequencies or velocities. If you specify frequencies, you can optionally specify a (new) reference code (see function setreferencecode for more details) and rest frequency (else the existing ones will be used).

If you specify velocities, you can optionally specify a (new) reference code and rest frequency (else the existing ones will be used). You must also give the doppler type (see function summary for more details). The velocities are then converted to frequency for creation of the Spectral Coordinate (which is fundamentally described by frequency).

You may specify the rest frequency as a Quantum or a double (native units of Spectral Coordinate used).

Example

```
- cs := coordsys(spectral=T);
-
- f1 := [1,1.01,1.03,1.4]
- fq := dq.quantity(f1, 'GHz')
- cs.setspectral(frequencies=fq)
- v := cs.frequencytovelocity(f1, 'GHz', 'radio', 'km/s')
- print 'v=', v
[88731.3175 86620.7061 82399.4832 4306.86125]
- vq := dq.quantity(v, 'km/s')
- cs.setspectral(velocities=vq, doppler='radio')
- f2 := cs.velocitytofrequency(v, 'GHz', 'radio', 'km/s')
-
- print 'f1 = ', f1
[1 1.01 1.03 1.4]
- print 'f2 = ', f2
[1 1.01 1.03 1.4]
```

We make a linear Spectral Coordinate. Then overwrite it with a list of frequenices. Convert those values to velocity, then overwrite the coordinate starting with a list of velocities. Then convert the velocities to frequency and show we get the original result.

---

coordsys.setstokes - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the Stokes types

Synopsis
setstokes(stokes)

Arguments

stokes	in	Stokes types Allowed:Vector of strings

Returns
T or fail

Description

If the Coordinate System contains a Stokes Coordinate, this function allows you to change the Stokes types defining it. If there is no Stokes Coordinate, an exception is generated.
See the coordsys constructor to see the possible Stokes types you can set.
You can set the Stokes types with function setstokes.

Example

```
- cs := coordsys(stokes='I V')
- cs.stokes()
I V
- cs.setstokes("XX RL")
- cs.stokes()
XX RL
```

---

coordsys.settabular - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set tabular values for the tabular coordinate

Synopsis
settabular(pixel, world, which)

Arguments

pixel	in	Vector of (1-rel) pixel values Allowed:Double vector Default: No change
world	in	Vector of world values Allowed:Double vector Default: No change
which	in	Which Tabular coordinate Allowed:Integer Default: 1

Returns
T or fail

Description

When you construct a Coordsys `tool`, if you include a Tabular Coordinate, it will be linear. This function allows you to replace the Tabular Coordinate by a finite table of values. Coordinate conversions between pixel and world are then done by interpolation (or extrapolation beyond the end). The table of values must be at least of length 2 or an exception will occur.
You may specify a vector of pixel and world values (in the current units of the Tabular Coordinate). These vectors must be the same length. If you leave one of them unset, then the old values are used, but again, ultimately, the pixel and world vectors must be the same length.
The new reference pixel will be the first pixel value. The new reference value will be the first world value.
Presently, there is no way for you to recover the lookup table once you have set it.
If you have more than one Tabular Coordinate, use argument `which` to specify which one you want to modify.

Example

```
- cs := coordsys(tabular=T);
- cs.settabular (pixel=[1,10,15,20,100], world=[10,20,50,100,500])
```

We make a linear Tabular Coordinate. Then overwrite it with a non-linear list of pixel and world values.

---

coordsys.settelescope - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set a new telescope

Synopsis
settelescope(value)

Arguments

value	in	New telescope Allowed:String

Returns
T or fail

Description

This function (short-hand name `st`) sets a new telescope of the observation. The telescope position may be needed for reference code conversions; this is why it is maintained in the Coordinate System. So it is fundamental to the Coordinate System and should be correct.
You can find a list of the observatory names know to AIPS++ with the Measures obslist function.
You can get the current telescope with function telescope.

Example

```
- cs := coordsys()
- cs.telescope()
ATCA
- cs.settelescope('VLA')
- cs.telescope()
VLA
-
- cs.settelescope('The One In My Backyard')
WARN: This telescope is not known to the AIPS++ system
You can request that it be added
-
- dm.obslist()
ALMA ATCA BIMA CLRO DRAO DWL GB GBT JCMT MOPRA NRAO12M PKS VLA WSRT
```

---

coordsys.setunits - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Set the axis units

Synopsis
setunits(value, type, overwrite, which)

Arguments

value	in	Units Allowed:Vector of strings
type	in	Which coordinate type ? Allowed:String from "direction", "stokes", "spectral", "linear", "tabular" or unset Default: All
overwrite	in	Overwrite linear or tabular coordinate units ? Allowed:Bool Default: F
which	in	Which coordinate if more than one of same type Allowed:Integer Default: First

Returns
T or fail

Description

Each axis associated with the Coordinate System has a unit. This function (short-hand name `su`) allows you to set new axis units.
You can set the units either for all axes (`type=unset`) or for just the axes associated with a particular coordinate type.
In general, the units must be consistent with the old units. When you change the units, the increment and reference value will be adjusted appropriately. However, for a linear or tabular coordinate, and only when you specify `type='linear'` or `type='tabular'` (i.e. you supply units only for the specified linear of tabular coordinate), and if you set `overwrite=T`, you can just overwrite the units with no further adjustments. Otherwise, the `overwrite` argument will be silently ignored. Use argument `which` to specify which coordinate if you have more than one of the specified type.
You can recover the current axis units with function units.

Example

```
- cs := coordsys(direction=T, spectral=T)
- cs.summary()
-
Direction reference : J2000
Spectral  reference : LSRK
Velocity  type      : RADIO
Rest frequency      : 1.42041e+09 Hz

Telescope        : ATCA
Observer         : Karl Jansky
Date observation : 2000/07/05/06:58:46

Axis Type      Name            Proj   Coord value at pixel    Coord incr Units
--------------------------------------------------------------------------------
1    Direction Right Ascension   SIN  00:00:00.000     1.00 -6.000000e+01 arcsec
2    Direction Declination      SIN +00.00.00.000     1.00  6.000000e+01 arcsec
3    Spectral  Frequency             1.415000e+09     1.00  1.000000e+03 Hz
               Velocity              1.140944e+03     1.00 -2.110611e-01 km/s
-
- cs.setunits(value="deg rad mHz");
-
-
Direction reference : J2000
Spectral  reference : LSRK
Velocity  type      : RADIO
Rest frequency      : 1.42041e+12 mHz

Telescope        : ATCA
Observer         : Karl Jansky
Date observation : 2000/07/05/06:58:46

Axis Type      Name            Proj   Coord value at pixel    Coord incr Units
--------------------------------------------------------------------------------
1    Direction Right Ascension   SIN  00:00:00.000     1.00 -6.000000e+01 arcsec
2    Direction Declination      SIN +00.00.00.000     1.00  6.000000e+01 arcsec
3    Spectral  Frequency             1.415000e+12     1.00  1.000000e+06 mHz
               Velocity              1.140944e+03     1.00 -2.110611e-01 km/s
```

---

coordsys.stokes - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the Stokes types

Synopsis
stokes()

Returns
Vector of strings or fail

Description

If the Coordinate System contains a Stokes Coordinate, this function recovers the Stokes types defining it. If there is no Stokes Coordinate, an exception is generated.
You can set the Stokes types with function setstokes.

Example

```
- cs := coordsys(stokes='I V')
- cs.stokes()
I V
```

---

coordsys.summary - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Summarize basic information about the Coordinate System

Synopsis
summary(doppler, list)

Arguments

doppler	in	List velocity information with this doppler definition Allowed:String Default: RADIO
list	in	List to global logger Allowed:Boolean Default: T

Returns
T, fail, or Vector of strings

Description

This function (short-hand name `s`) summarizes the information contained in the Coordinate System.
For spectral coordinates, the information is listed as a velocity as well as a frequency. The argument `doppler` allows you to specify what doppler convention it is listed in. You can choose from `radio, optical` and `beta`. Alternative names are `z` for `optical`, and `relativistic` for `beta`. The default is `radio`. The definitions are

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

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 with the function setrestfrequency.
These velocity definitions are provided by the measures system via the Doppler measure (see example).
If you set `list=F`, then the summary will not be written to the global logger. However, the return value will be a vector of strings holding the summary information, one string per line of the summary.
For direction and spectral coordinates, the reference frame (e.g. J2000 or LSRK) is also listed. Along side this, in parentheses, will be the conversion reference frame as well (if it is different from the native reference frame). See function setconversion to see what this means.

Example

```
- d := dm.doppler('beta')
- dm.listcodes(d)
[normal=RADIO Z RATIO BETA GAMMA OPTICAL TRUE RELATIVISTIC, extra=]
```

```
-
- cs := coordsys(direction=T, spectral=T)
- cs.summary()
-
Direction reference : J2000
Spectral  reference : LSRK (-> BARY)
Velocity  type      : RADIO
Rest frequency      : 1.42041e+09 Hz

Telescope        : ATCA
Observer         : Karl Jansky
Date observation : 2000/07/04/10:01:11

Axis Type      Name             Proj   Coord value at pixel    Coord incr Units
-------------------------------------------------------------------------------
1    Direction Right Ascension   SIN  00:00:00.000     1.00 -6.000000e+01 arcsec
2    Direction Declination       SIN +00.00.00.000     1.00  6.000000e+01 arcsec
3    Spectral  Frequency              1.415000e+09     1.00  1.000000e+03 Hz
               Velocity               1.140944e+03     1.00 -2.110611e-01 km/s
```

---

coordsys.telescope - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Return the telescope

Synopsis
telescope(measure)

Arguments

measure	in	Return as position measure Allowed:T or F Default: F

Returns
String or position measure or fail

Description

This function (short-hand name `t`) returns the telescope contained in the Coordinate System. It can be returned as a simple string (`measure=F`) or as a position measure (`measure=T`).
The telescope position may be needed for reference code conversions; this is why it is maintained in the Coordinate System.
The conversion from string to position is done with observatory. The example shows you what is really going on.

Example

```
- cs := coordsys()
- cs.telescope()
ATCA
- cs.telescope(T)
[type=position, refer=ITRF, m2=[value=6372960.26, unit=m],
m1=[unit=rad, value=-0.52613792], m0=[unit=rad, value=2.61014232]]
-
- dm.observatory(cs.telescope())
[type=position, refer=ITRF, m2=[value=6372960.26, unit=m],
m1=[unit=rad, value=-0.52613792], m0=[unit=rad, value=2.61014232]]
```

We get the telescope as a string and as a position measure. We also show how the Measures system was used to convert from the simple name to a position Measure.

---

coordsys.toabs - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert relative coordinate to absolute

Synopsis
toabs(value, isworld)

Arguments

value	in	Relative coordinate Allowed: Numeric vector, vector string, vector quantum, record
isworld	in	Is coordinate world or pixel ? Allowed: T, F, unset Default: unset

Returns
Same as input or fail

Description

This function converts a relative coordinate to an absolute coordinate. The coordinate may be a pixel coordinate or a world coordinate.
If the coordinate is a pixel coordinate, it is supplied as a numeric vector. If the coordinate is a world coordinate, you may give it in all of the formats described in the formatting discussion.
If the coordinate value is supplied by a Coordsys `tool` function (e.g. toworld) then the coordinate 'knows' whether it is world or pixel (and absolute or relative). However, you might supply the value from some other source as a numeric vector (which could be world or pixel) in which case you must specify whether it is a world or pixel coordinate via the `isworld` argument.

Example

```
- cs := coordsys(direction=T, spectral=T)
-
- aw := cs.toworld([100,100,24], 's')
- rw := cs.torel(aw)
- aw2 := cs.toabs(rw)
-
- aw
23:53:23.781 +01.39.00.821 1.415023e+09 Hz
- rw
5.943285e+03 arcsec 5.940821e+03 arcsec 2.300000e+04 Hz
- aw2
23:53:23.781 +01.39.00.821 1.415023e+09 Hz
```

This example uses world coordinates.

Example

```
- cs := coordsys(direction=T, spectral=T)
-
- ap := cs.topixel()           # Reference value
- rp := cs.torel(ap)
- ap2 := cs.toabs(rp)
-
- ap
[1 1 1]
- rp
[0 0 0]
- ap2
[1 1 1]
```

This example uses pixel coordinates.

---

coordsys.toabsmany - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert many numeric relative coordinates to absolute

Synopsis
toabsmany(value, isworld)

Arguments

value	in	Relative coordinates Allowed:Numeric matrix
isworld	in	Is coordinate world or pixel ? Allowed:T, F, unset Default: unset

Returns
Numeric matrix or fail

Description

This function converts many relative coordinates to absolute. It exists so you can efficiently make many conversions (which would be rather slow if you did

them all with toabs). Because speed is the object, the interface is purely in terms of numeric matrices, rather than being able to accept strings and quanta etc. like toabs can.
When dealing with world coordinates, the units of the numeric values must be the native units, given by function units.

Example

```
- cs := coordsys(direction=T, spectral=T)    # 3 axes
-
- rv := cs.referencevalue();                 # reference value
- w := cs.torel(rv)                          # make relative
- inc := cs.increment();                     # increment
- idx := 1:100;
- off := inc[3] * idx;                       # offset for third axis
- wrel := array (w, 3, 100);                 # 100 conversions each of length 3
- wrel[3,] +:= off;                          # Make spectral axis values change
-
- wabs  := cs.toabsmany (rel, T)             # Convert
- wabs[,1]                                   # First absolute coordinate
[0 0 1.415001e+09]
- wabs[,100]                                 # 100th absolute coordinate
[0 0 1.415002e+09]
```

This example uses world coordinates.

coordsys.topixel - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert from absolute world to pixel coordinate

Synopsis
topixel(value)

Arguments

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

Returns
Vector of double or fail

Description

This function converts between world (physical) coordinate and absolute pixel coordinate (1-rel).
The world coordinate can be provided in one of four formats via the argument `world`. These match the output formats of function toworld.
If you supply fewer world values than there are axes in the Coordinate System, your coordinate vector will be padded out with the reference value for the missing axes. Excess values will be silently ignored.
You may supply the world coordinate in all of the formats described in the formatting discussion.

Example

```
- cs := coordsys(direction=T, spectral=T, stokes="I V", linear=2)
-
- w := cs.toworld([-2,2,1,2,23,24], 'n')
- cs.topixel(w)
[-2 2 1 2 23 24]
-
- w := cs.toworld([-2,2,1,2,23,24], 'q')
- cs.topixel(w)
[-2 2 1 2 23 24]
-
- w := cs.toworld([-2,2,1,2,23,24], 'm')
- cs.topixel(w)
[-2 2 1 2 23 24]
-
- w := cs.toworld([-2,2,1,2,23,24], 's')
- cs.topixel(w)
[-2 2 1 2 23 24]
-
- w := cs.toworld([-2,2,1,2,23,24], 'mnq')
- cs.topixel(w)
```

```
[-2 2 1 2 23 24]
```

Example

```
- cs := coordsys (stokes="I V", linear=2)
- cs.toworld([1,2,3], 's')
I 1.000000e+00 2.000000e+00
-
- cs.toworld([1,2,3], 'm')
[stokes=I, linear=[*1=[value=1, unit=km], *2=[value=2, unit=km]]]
-
- cs.toworld([1,2,3], 'q')
[*1=[value=1, unit=], *2=[value=1, unit=km], *3=[value=2, unit=km]]
```

Example

```
- cs := coordsys (spectral=T, linear=1)
- cs.toworld([1,2,3], 'q')
[*1=[value=1.415e+09, unit=Hz], *2=[value=1, unit=km]]
```

---

coordsys.topixelmany - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert many absolute numeric world coordinates to pixel

Synopsis
topixelmany(value)

Arguments

value	in	Absolute world coordinates Allowed:Numeric matrix

Returns
Numeric matrix or fail

Description

This function converts many absolute world coordinates to pixel coordinates. It exists so you can efficiently make many conversions (which would be rather slow if you did them all with topixel). Because speed is the object, the interface is purely in terms of numeric matrices, rather than being able to accept strings and quanta etc. like topixel can.
The units of the numeric values must be the native units, given by function units.

Example

```
- cs := coordsys(direction=T, spectral=T)    # 3 axes
-
- rv := cs.referencevalue();                 # reference value
- inc := cs.increment();                     # increment
- idx := 0:99;
- off := inc[3] * idx;                       # offset for third axis
-
- wabs := array (rv, 3, 100);                # 100 conversions each of length 3
- wabs[3,] +:= off;                          # Make spectral axis values change
-
- pabs  := cs.topixelmany (wabs);            # Convert
- pabs[,1]                                   # First absolute pixel coordinate
[1 1 1]
- pabs[,100]                                 # 100th absolute pixel coordinate
[1 1 100]
```

coordsys.torecord - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert Coordinate System to a record

Synopsis
torecord()

Returns
Glish record or fail

Description

You can convert a Coordinate System to a record with this function. There is also fromrecord to set a Coordinate System from a record.
These functions are not for general user use; they are used for sending Coordsys `tools` about the *Glish* bus.

Example

```
- cs := coordsys(direction=T, stokes="I Q")
- rec := cs.torecord();
- cs2 := coordsys();
- cs2.fromrecord(rec);
-
- print cs.ncoordinates(), cs2.ncoordinates()
2 2
```

---

coordsys.torel - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert absolute coordinate to relative

Synopsis
torel(value, isworld)

Arguments

value	in	Absolute coordinate Allowed: Numeric vector, vector string, vector quantum, record
isworld	in	Is coordinate world or pixel ? Allowed: T, F, unset Default: unset

Returns
Same as input or fail

Description

This function converts an absolute coordinate to a relative coordinate. The coordinate may be a pixel coordinate or a world coordinate.
Relative coordinates are relative to the reference pixel (pixel coordinates) or the reference value (world coordinates) in the sense
$relative = absolute - reference$.
If the coordinate is a pixel coordinate, it is supplied as a numeric vector. If the coordinate is a world coordinate, you may give it in all of the formats described in the formatting discussion.
If the coordinate value is supplied by a Coordsys `tool` function (e.g. toworld) then the coordinate 'knows' whether it is world or pixel (and absolute or relative). However, you might supply the value from some other source as a numeric vector (which could be world or pixel) in which case you must specify whether it is a world or pixel coordinate via the `isworld` argument.

Example

```
- cs := coordsys(direction=T, spectral=T)
-
- aw := cs.toworld([100,100,24], 's')
- rw := cs.torel(aw)
- aw2 := cs.toabs(rw)
-
- aw
23:53:23.781 +01.39.00.821 1.415023e+09 Hz
- rw
5.943285e+03 arcsec 5.940821e+03 arcsec 2.300000e+04 Hz
- aw2
23:53:23.781 +01.39.00.821 1.415023e+09 Hz
```

This example uses world coordinates.

Example

```
- cs := coordsys(direction=T, spectral=T)
-
- ap := cs.topixel()           # Reference value
```

```
- rp := cs.torel(ap)
- ap2 := cs.toabs(rp)
-
- ap
[1 1 1]
- rp
[0 0 0]
- ap2
[1 1 1]
```

This example uses pixel coordinates.

---

coordsys.torelmany - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert many numeric absolute coordinates to relative

Synopsis
torelmany(value, isworld)

Arguments

value	in	Absolute coordinates Allowed:Numeric matrix
isworld	in	Is coordinate world or pixel ? Allowed:T, F, unset Default: unset

Returns
Numeric matrix or fail

Description

This function converts many absolute coordinates to relative. It exists so you can efficiently make many conversions (which would be rather slow if you did them all with torel). Because speed is the object, the interface is purely in terms of numeric matrices, rather than being able to accept strings and quanta etc. like torel can.

When dealing with world coordinates, the units of the numeric values must be the native units, given by function units.

Example

```
- cs := coordsys(direction=T, spectral=T)    # 3 axes
-
- w := cs.referencevalue();                  # reference value
- inc := cs.increment();                     # increment
- idx := 0:99;
- off := inc[3] * idx;                       # offset for third axis
- wabs := array (w, 3, 100);                 # 100 conversions each of length 3
- wabs[3,] +:= off;                          # Make spectral axis values change
-
- wrel  := cs.torelmany (wabs, T)            # Convert
- wrel[,1]                                   # First relative coordinate
[0 0 0]
- wrel[,100]                                 # 100th relative coordinate
[0 0 99000]
```

This example uses world coordinates.

coordsys.toworld - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert from absolute pixel coordinate to world

Synopsis
toworld(value, format)

Arguments

value	in	Absolute pixel coordinate Allowed:Numeric vector Default: Reference pixel
format	in	Format string Allowed:String from combination of "n", "q", "s", "m" Default: 'n'

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

Description

This function converts between absolute pixel coordinate (1-rel) and absolute world (physical coordinate).
If you supply fewer pixel values than there are axes in the Coordinate System, your coordinate vector will be padded out with the reference pixel for the missing axes. Excess values will be silently ignored.
You may ask for the world coordinate in all of the formats described in the discussion regarding the formatting possibilities available via argument `format`.

Example

```
- cs := coordsys(direction=T, spectral=T)
-
- cs.toworld([-2,2,2], 'n')
[3.00000051 1.00000001 1.415001e+09]
-
- cs.toworld([-2,2,2], 'q')
[*1=[value=3.00000051, unit='],
 *2=[value=1.00000001, unit='],
 *3=[value=1.415001e+09, unit=Hz]]
-
- cs.toworld([-2,2,2], 'm')
[direction=[type=direction, refer=J2000, m1=[value=0.000290888213, unit=rad],
                                         m0=[unit=rad, value=0.000872664774]],
 spectral=[frequency=[type=frequency, refer=LSRK, m0=[value=1.415001e+09, unit=Hz]],
           radiovelocity=[value=1140.73308, unit=km/s],
           opticalvelocity=[value=1145.09023, unit=km/s],
           betavelocity=[value=1142.90335, unit=km/s]]]
-
- cs.toworld([-2,2,2], 's')
00:00:12.000 +00.01.00.000 1.415001e+09
```

Example

```
- cs := coordsys (stokes="I V", linear=2)
- cs.toworld([1,2,3], 's')
I 1.000000e+00 2.000000e+00
```

```
-
- cs.toworld([1,2,3], 'm')
[stokes=I, linear=[*1=[value=1, unit=km], *2=[value=2, unit=km]]]
-
- cs.toworld([1,2,3], 'q')
[*1=[value=1, unit=], *2=[value=1, unit=km], *3=[value=2, unit=km]]
```

Example

```
- cs := coordsys (spectral=T, linear=1)
- cs.toworld([1,2,3], 'q')
[*1=[value=1.415e+09, unit=Hz], *2=[value=1, unit=km]]
```

---

coordsys.toworldmany - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert many absolute pixel coordinates to numeric world

Synopsis
toworldmany(value)

Arguments

value	in	Absolute pixel coordinates Allowed:Numeric matrix

Returns
Numeric matrix or fail

Description

This function converts many absolute pixel coordinates to world coordinates. It exists so you can efficiently make many conversions (which would be rather slow if you did them all with toworld). Because speed is the object, the

interface is purely in terms of numeric matrices, rather than being able to produce strings and quanta etc. like toworld can.
The units of the output world values are the native units given by function units.

Example

```
- cs := coordsys(direction=T, spectral=T)    # 3 axes
-
- rp := cs.referencepixel();                 # reference pixel
- off := 0:99;                               # offset for third axis
-
- pabs := array (rp, 3, 100);                # 100 conversions each of length 3
- pabs[3,] +:= off;                          # Make spectral axis values change
-
- wabs  := cs.toworldmany (pabs);            # Convert
- wabs[,1]                                   # First absolute pixel coordinate
[0 0 1.415e+09]
- wabs[,100]                                 # 100th absolute pixel coordinate
[0 0 1.415099e+09]
```

coordsys.type - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Return the type of this tool

Synopsis
type()

Returns
String

Description

This function returns the string 'coordsys'.

---

coordsys.units - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Recover the units for each axis

Synopsis
units(type)

Arguments

type	in	Coordinate type Allowed: String from 'direction', 'stokes', 'spectral', 'linear' or unset Default: All

Returns
Vector of strings or fail

Description

Each axis associated with the Coordinate System has a unit. This function (short-hand name `u`) returns those units (in world axis order).
You can recover the units either for all coordinates (leave `type` unset) or for a specific coordinate type (mimumum match of the allowed types will do). If you ask for a non-existent coordinate an exception is generated.
You can set the units with function setunits.

Example

```
- cs := coordsys(direction=T, spectral=T)
- cs.units(T)
' ' Hz
-
- cs.units('spec')
Hz
```

---

coordsys.velocitytofrequency - Function

Package	**general**
Module	**images**
Tool	**coordsys**

Convert velocity to frequency

Synopsis
velocitytofrequency(value, frequnit, doppler, velunit)

Arguments

value	in	Velocity to convert Allowed:Numeric or vector of numeric
frequnit	in	Unit of output frequencies Allowed:String Default: Intrinsic units
doppler	in	Velocity doppler definition Allowed:String Default: 'radio'
velunit	in	Unit of input velocities Allowed:String Default: 'km/s'

Returns
Double or vector of double or fail

Description

This function (short-hand name **vtf**) converts velocities to frequencies.
The input velocities are specified via a vector of numeric values, a specified unit (`velunit`), and a velocity doppler definition (`doppler`).
The frequencies are returned in a vector for which you specify the units (`frequnit`). If you don't give the unit, it is assumed that the units are those given by function units for the spectral coordinate.
This function will return a fail if there is no spectral coordinate in the Coordinate System. See also the function frequencytovelocity.

Example

```
- im := image('hcn.cube')
- cs := im.coordsys()
-
- local pa, wa;
- cs.findcoordinate(pa, wa, 'spectral')     # Find spectral axis
- pixel := cs.referencepixel();             # Use reference pixel for non-spectral
-
- nFreq := im.shape()[pa[1]];               # Length of spectral axis
- freq := [];
- for (i in 1:nFreq) {
    pixel[pa[1]] := i;                      # Assign value for spectral axis of pixel coordinate
    w := cs.toworld(value=pixel, format='n')# Convert pixel to world
    freq[i] := w[wa[1]];                    # Fish out frequency
  }
- vel := cs.frequencytovelocity(value=freq, doppler='optical', velunit='km/s')
- freq2 := cs.velocitytofrequency(value=vel, doppler='optical', velunit='km/s')
```

In this example, we find the optical velocity in km/s of every pixel along the spectral axis of our image. First we obtain the Coordinate System from the image. Then we find which axis of the Coordinate System (image) pertain to the spectral coordinate. Then we loop over each pixel of the spectral axis, and convert a pixel coordinate (one for each axis of the image) to world. We obtain the value for the spectral axis from that world vector, and add it to the vector of frequencies. Then we convert that vector of frequencies to velocity. Then we convert it back to frequency. They better agree.

1.2.11 is_coordsys - Function

Package general
Module images

Is this variable a Coordsys tool ?

Synopsis
is_coordsys(thing)

Arguments

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

Returns
T or F

Description

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

Example

```
cs := coordsys();
is_coordsys(cs)
T
cs2 := 'doggies'
is_coordsys(cs2)
F
```

1.2.12 coordsystools - Function

Package general
Module images

Find the list of current global Coordsys tools

Synopsis
coordsystools()

Returns
Vector of Strings

Description
This function retuns a list of the names of the Coordsys `tools` currently in Global name space.

Example

```
cs1 := coordsys();
im := shapetoimage(shape=[10,10])
cs2 := im.coordsys();
coordsystools()
cs1 cs2
```

1.2.13 coordsystest - Function

Package general
Module images

Run test suite for Coordsys tool

Synopsis
coordsystest(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 Coordsys `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). Note that one of the tests is if the created coordinate system tools match the actual list of tools. If a coordinate system has been created outside the test, the tests will fail. The `coordsystools` function can be used to check for pre-existing tools. They can be removed by re-assigning them. E.g.:

```
cs := T
```

Example

```
coordsystest()
```

Runs all tests.

Example

```
coordsys(4)
```

Runs only test 4.

1.2.14 imagefitter - Tool

Package general
Module images

Fit models to images of the sky
include "imagefitter.g"

Constructors

imagefitter	Construct an Imagefitter

Functions

componentlist	Recover the fitted components
done	Destroy this Imagefitter tool
gui	Start the GUI interface
nfits	Return number of accepted fits
regions	Recover fitted regions
setmaxpixels	Set maximum number of pixels to fit without query
summary	Summarize the image being fitted
type	What type is this tool ?

Synopsis
The Imagefitter `tool` offers interactive fitting of 2D models (or components) to an image of the sky. You control the process from a GUI interface. The `tool` is self contained. Thus, the display is built into the interface.
Current functionality includes

- The image is displayed with the Viewer providing full control over the display (zooming, colour tables, transfer function, labels etc) and easy saving to postscript of the display.

- The basic interface allows only Gaussian fits of single components with an automatic estimate. A more advanced interface allows simultaneous fits of multiple components with interactive or automatic estimates.
- Choice of models with the advanced interface.
- Model parameters can be held fixed in the fitting process with the advanced interface.
- Interactive specification of `regions-of-interest` (box or polygon) to be fitted
- Accepted fits are stored and returned Componentlist tools via function componentlist. The fitted components can be optionally deconvolved from the restoring beam in this step. The Componentlist can be used as a model in imager.
- The `regions-of-interest` used for the accepted fits are stored internally and can be recovered.
- Fit parameters are listed and the fits can be graphically overlaid on the display.
- Fit residuals are displayed via an image, histogram and statistics
- The fitted model can be subtracted/added to the image
- `Regions-of-interest` can be recovered from a Table and automatically refit.
- You can step through sky planes of a multidimensional image fitting as you go.

Restrictions which will be removed with time are currently

- Only Gaussian and Point (a Gaussian of the same shape as the restoring beam) models are available
- There is no joint Stokes fitting
- There is no automatic source finding

There will be some interface changes as these things are accomodated.

More Details
It is useful for you to know a little about how the Imagefitter manages the resultant fitted parameters. Each 'fit' that you make may contain multiple components. For example, you might fit three gaussians simultaneously. The Imagefitter stores the result of each accepted fit in its own Componentlist `tool`. In reality it maintains a list of Componentlist tools and you can

manipulate this list in a few ways. You can display (make graphical overlay on display) any or all of the Componentlist tools in the list (via the Show/Hide buttons on the main GUI). You may delete any or all of the Componentlist tools in the list (via the Delete button on the main GUI). You can recover all or any of the Componentlist tools in the list (via function componentlist).
Here is an example of the basic Imagefitter GUI.
Under the “File” menu at the top you find the following items.

- Save regions : when a fit is accepted, the region that was fit is stored as well. You can save those regions to a Table if you wish (this invokes the Saveregions widget).
- Restore regions and fit : this enables you to recover previously stored regions from a Table and automatically fit them (single Gaussian component only).
- Dismiss : dismiss the GUI (you can recover it with the gui function).
- Done : destroy the Imagefitter tool

The GUI shows a display of the image to the right. Beneath the image is a listbox showing the parameters of the model fits. To the left are a range of displays for the residuals of the fits. At the top is a histogram and some statistics. Below that is an image of the residuals in the fitted region.
There are two rollups beneath the data displays.
The first is labelled “Pixel Selection”. With this widget, you can select which pixels, by intensity range, will be fit - it is generally good to exclude noise pixels. You can set an inclusion range, an exclusion range, or use an auto pixel selection. This last mode selects pixels depending on the context. For Stokes I, positive pixels are selected if the absolute peak in the fitting region is positive. It selects negative pixels if the peak is negative. For other Stokes parameters, all pixels are selected. You may wish to fine tune the pixel selection range so that only pixels above the noise level are selected.
Also in this rollup is an entry box labelled ’Maximum pixels’. If you ask to fit a large number of pixels (perhaps you forgot to set a region), the fit may go for a semi-infinite time. To protect against this, there is a built in maximum number of pixels allowed. If you ask to fit more pixels than this, you are queried whether you wish to proceed through a choice GUI. You can set this maximum pixels query limit to something different here (perhaps make it larger).
The second rollup is labelled “Fine Control”. This is where you can activate the advanced interface. You select the number of simultaneous components you wish to fit (either type it in or move the slider) and then press the “Go” button.
Finally, a message line below the rollups provides you with information on what to do next. The Imagefitter constrains you as much as it can so that the only things you can do are valid.

The Basic Interface
If you don't activate the advanced interface, you get the basic interface. This means that

- Only one Gaussian component is fitted
- You cannot fix any parameters of the fit
- The intitial estimate is automatic

The fitting process is triggered by you generating a region (rectangle or polygon) and double clicking within it to signify it is ready. After the fit is done, the parameters are displayed in the list box, and the residual image, histogram and statistics are displayed. At this point, you can press the "Accept" button to accept and store the fit internally. Otherwise you can just do another fit.
At any point you can press the "Subtract" button. This subtracts the last fitted model from the data and the main image display is updated to reflect this. Once the "Subtract" button is pressed, it becomes an "Add" button, as there is not much point to you subtracting a model more than once, fun though it might be.
The fit parameters are displayed in the listbox under the main display. Here you may select the current fit or any of the previously accepted fits from the list. Its components (a fit may have multiple components in the advanced interface) will be listed. You may also show/hide graphically the selected fit on the display. You may also delete fits from the list if you decide you erroneously accepted a fit earlier.

The Advanced Interface
If you activate the advanced interface (via the "Fine Control" rollup and the "Go" button), then

- You can fit many simultaneous components
- In principle those components can be different types, but currently only types Gaussian and Point (a Gaussian with the same shape as the restoring beam) are implemented.
- You can fix some parameters of the fit
- The intitial estimate can be interactive or automatic

With the advanced interface, the order in which you do things is not as linear as with the basic interface. However, the idea is that you first establish an estimate for each component, and then do the fit.
Here is an example of the advanced Imagefitter GUI with three components available.
You select a particular component by clicking on the numbered (TAB) button at the top of the GUI (this is a pseudo-tab widget interface). This brings to

the front a GUI for this particular component. You then choose the type of component you wish to fit (optionmenu in the top left). Then you establish an estimate for each component and then fit them simultaneously in a specified region.

At the bottom of the advanced GUI, there is a Region widget. This houses any region you create interactively with the main display (via the rectangle or polygon region maker). This region is used in two ways. Firstly, this is the region that is used when you press the "Fit" button. Secondly, when you make an automatic estimate for a particular component, the pixels are only selected from this region. Thirdly, when you make an automatic estimate for all components simultaneously, the region must embrace all of the sources of interest (it would be the same region as when doing the actual fit). Thus the region has a context dependent use - one component specific (auto individual component estimate) and one global (the fit and auto estimate for all components). If you have not explicitly set a region, it defaults to the unzoomed full region currently being displayed.

To aid you in using this Region widget, there is a check button (labelled "Inserted") which is set on for half a second when a region is captured from the main display. There is also a button labelled "Show". This lists the bounding box of the region and redisplay the region on the image.

Now the main thing you must do is establish an estimate for that component. You can do this in a variety of ways. Firstly, you can enter manually the values for each parameter. Secondly, you can press the "Auto" button and an automatic estimate will be made for you. Thirdly, you can press the "Inter" button and use the displayed overlay to establish an interactive estimate.

It is also possible to make automatic estimates for *all* of the components (rather than one by one). You do this by establishing a region that embraces all of the components. Then press the "Est" button at the bottom and it will make an estimate for each component requested. It won't work terribly well if the components are heavily overlapping.

Once you are happy with the estimates for each component, you press the "Fit" button. You can then click the "Accept" button on the main GUI display (as in the Basic interface) if you want to keep the fit, or you can just do another one.

The buttons labelled "Fixed" for each parameter enable you to fix that parameter at its estimated value during the fit. This must be an active value to be useful. To aid in setting an estimated value which you wish to fix, the left-pointing arrows transfer the values from the fit entry boxes to the estimate boxes.

The "Pixel Selection" rollup is active for the Advanced as well as the Basic interface.

Finally please note that for your convenience, the flux density is presented to you by the Imagefitter GUI interfaces as a peak flux density. Inside the Componentlist, the flux density is stored in an Integral representation. Therefore conversions back and forth go on behind the scenes.

Events
The Imagefitter emits one event called 'accept'. Whenever the user presses the 'Accept' button to accept a fit, this event is emitted. The value of the event is the fit number.

imagefitter - Constructor

Package	**general**
Module	**images**
Tool	**imagefitter**

Construct an Imagefitter

Synopsis
imagefitter(infile, region, parent, residual, maxpix, widgetset)

Arguments

infile	in	The image file name or image tool Allowed:The file name or an Image tool
region	in	The region of interest to display Allowed:Region tool Default: Whole image
parent	in	Parent frame to embed GUI in Allowed:Tk frame agent Default: F
residual	in	Residual image file name Allowed:String Default: Temporary image
maxpix	in	Maximum number of pixels to fit without query Allowed:Integer Default: 256*256
widgetset	in	The widgetset to use for the GUI Allowed:A widgetserver tool Default: ddlws

Returns
imagefitter tool

Description

This is the constructor. You must specify the `image file` either by its name or by giving an `image tool`. You may optionally embed the display in an existing Tk frame (`parent`) if you wish. This constructor does a fair amount of work, so be a little patient if need be.
The input image is copied (and any specified region applied); this copied image is used should you choose to subtract any model fits (via the GUI). If you specify the argument `residual`, then the copied image is saved in a file of this name. If you leave `residual` unspecified, a temporary image will be created and used for model subtractions, but it will be destroyed when you terminate the Imagefitter.
See the description at the top of this document to find details on operating the Imagefitter interface.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- myfitter.done()
- im := image('ngc1313.cube')
- myfitter := imagefitter(infile=im, residual='ngc1313.resid')
```

These examples show the two ways to use the `infile` argument. In the first case we indicate that should we subtract models from the image, that they should be subtracted from the `image file` called *ngc1313.cube*. In the second case we indicate that should we subtract models from the image, that the input image is not changed, and the residuals are written to the `image file` called *ngc1313.resid*.

---

imagefitter.componentlist - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

Recover the fitted components

Synopsis
componentlist(which, concatenate, deconvolve)

Arguments

which	in	Which fit ? Allowed:Unset or integer Default: All
concatenate	in	Concatenate all Componentlists into one ? Allowed:T or F Default: F
deconvolve	in	Deconvolve components from restoring beam Allowed:T or F Default: F

Returns

Record of Componentlists, Componentlist or fail

Description

The Imagefitter stores the fitted Sky components in an internal record. For each accepted fit, there is one entry in the record (indexed 1, 2, 3 ...) and each entry holds a Componentlist `tool`; there may be any number of components in each Componentlist (depends upon how many components you fitted). If there are no accepted fits, it returns an empty record and issues a warning.
By default, this function returns the record holding all of the separate Componentlists. If you do specify `which` (the integer index of the accepted fit) then it returns the Componentlist `tool` for that index directly.
If you leave `which` unset (its default), but you specify `concatenate=T`, then all the Componentlists are concatenated into one, and that concatenated Componentlist `tool` is returned directly. The `concatenate` argument is meaningless if you do specify a particular index with `which` (see above).
You may also optionally deconvolve the components from the restoring beam of the image if there is one. The default is not to do this.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- list := myfitter.componentlist()                    # Recovers record of Componentlist tools
- print list[1].length()                              # Length (number components) of
3                                                     # first Componentlist
#
- cl := myfitter.componentlist(which=3)               # Recovers specific Componentlist
- print cl.length()
2
-
- cld := myfitter.componentlist(which=2, deconvolve=T)    # FIt 2 and deconvolve from beam
```

imagefitter.done - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

Destroy this Imagefitter tool

Synopsis
done()

Returns
Bool or fail

Description
This function destroys the Imagefitter `tool`.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- myfitter.done()
- myfitter
F
```

---

imagefitter.gui - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

Start the GUI interface

Synopsis
gui()

Returns
Bool or fail

Description
If you dismiss the GUI (with the dismiss button), then you can recover it with this function. The state of the Imagefitter and the GUI is preserved. If the GUI is already present, then it does nothing.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- myfitter.gui()
```

---

imagefitter.done - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

Destroy this Imagefitter tool

Synopsis
done()

Returns
Bool or fail

Description
This function destroys the Imagefitter `tool`.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- myfitter.done()
- myfitter
F
```

---

imagefitter.nfits - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

Return number of accepted fits

Synopsis
nfits()

Returns
Integer

Description

This function retuns the number of fits that the user has accepted.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- print 'Number of fits = ', myfitter.nfits()
```

---

imagefitter.regions - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

Recover fitted regions

Synopsis
regions(which)

Arguments

which	in	Which fit ? Allowed:Unset or integer Default: All

Returns
Record of regions, region, or fail

Description

As well as the internal fitlist (a record of Componentlist `tools`), there is also a record of regions, corresponding to the accepted fits that is stored internally. This function recovers those regions. By default, you get them all in an indexed record. You can specify a specific one if you wish, and then the function returns the region directly (rather than putting it in a record).

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- r := myfitter.regions()
- print 'number of fitted regions=', length(r)
- im := image('ngc1313.cube')
- im.view(region=r[1])
```

Here the region of the image that was fitted for the first component is displayed.

---

imagefitter.setmaxpixels - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

Set maximum number of pixels to fit without query

Synopsis

setmaxpixels(maxpix)

Arguments

maxpix	in	Number of pixels

Returns

T

Description

If you ask to fit a large number of pixels (perhaps you forgot to set a region), the fit may go for a semi-infinite time. To protect against this, there is a built in maximum number of pixels allowed. If you ask to fit more pixels than this, you are queried whether you wish to proceed through a choice GUI.
This function lets you set the maximum pixels value.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube', maxpixels=512*512)
- myfitter.setmaxpixels(maxpixles=128*128)                          # My computer is slower than I thought
```

---

imagefitter.summary - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

Summarize the image being fitted

Synopsis
summary()

Returns
T or fail

Description
This function summarizes, to the logger, the image with which the fitter was constructed. It just invokes the Image `tool` function summary.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- myfitter.summary()
```

---

imagefitter.type - Function

Package	**general**
Module	**images**
Tool	**imagefitter**

What type is this tool ?

Synopsis
type()

Returns
String or fail

Description
Returns the string “imagefitter”.

Example

```
- myfitter := imagefitter(infile='ngc1313.cube')
- myfitter.type()
imagefitter
```

1.2.15 imagefittertest - Function

Package general
Module images

Run test suite for Imagefitter tool

Synopsis
imagefittertest(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 Imagefitter `tool`. This function is very limited presently as it's hard to test an interactive application.
To run a specific test, you can use the `which` argument. Since the number of tests keeps growing, we don't tell you how many there are here or what exactly they do ! If you give too large a value, nothing will happen. `which` can be an integer or a vector of integers.

Example

```
- imagefittertest()
```

Runs all tests.

1.2.16 imageprofilefitter - Tool

Package general
Module images

Fit models to 1D profiles from images
include "imageprofilefitter.g"

Constructors

imageprofilefitter	Construct an imageprofilefitter

Functions

done	Destroy this imageprofilefitter tool
getestimate	Recover the estimate
getfit	Recover the fit
getprofiles	Get data, estimate and fit as vectors
getstore	Recover all fits added to internal store
gui	Start the GUI interface
setimage	Set a new image
type	What type is this tool ?

The imageprofilefitter `tool` offers interactive fitting of 1D models (or components) to profiles from an image. You control the process from a GUI interface. The `tool` is self contained so that the display is built into the interface.
You may provide one or two images (primary in keyword `infile` and

secondary in keyword `infile2`). If you only provide the primary image, it must have at least 3 dimensions. Then, a zeroth-order moment secondary image (simple average of all pixels along profile with 3-sigma clip) is automatically made for you along the specified profile axis (and removed when you destory the Imageprofilefitter tool or exit from Glish). This secondary image is then displayed and from it you interactively specify profiles to fit. If you provide the secondary image as well, it could be a pre-computed moment image (you might not like the automatically provided one). The secondary image does not actually have to be generated from the primary It is only its world coordinates that are used. For example, it could be an optical image from which you specify a direction and the profile is then extracted from the primary image (which might be a spectral-line neutral Hydrogen cube).

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube', axis=3)
```

Specify just the primary image.

Example

```
- pf2 := imageprofilefitter(infile='ngc1313.cube', infile2='ngc1313.mom', axis=3)
Specify the primary and secondary images.
```

The display of the secondary image looks like:
There are now two ways to drive the fitting process. You can either interactively fit individual profiles one at a time, or you can automatically fit all profiles in a selected region.

Individual Profiles
From the Viewer control buttons (buttons to the left of the display of the image), select the position handling button. Now go to the image display, position the cursor somewhere and double click. This will cause a second GUI to appear as below.
The profile is extracted from the nearest pixel and plotted in the secondary GUI. If you wish, the profile can be spatially averaged, rather than just being a single-pixel profile. Just enter the half-width (in pixels) in the entry box of the main GUI (defaults to 0; no averaging). The title of the plot includes the world and center pixel coordinate of the profile.
A profile can also be generated by selecting one of the region generation tools (rectangle or polygon) from the Viewer control buttons. If the Region Usage menu has 'Average' selected, then the profile will be spatially averaged over the region. Generate the region on the image display by dragging out the box or marking the polygon. As always, double click within the region to emit it.

There are various buttons and menus beneath the plot of the profile. You can select different x-axis units, absolute or relative units etc. From here the process is simple. You select the number of components you want to fit with the slider. For each component, a parameters panel is generated (you access different components with the TAB-widget buttons). Then for each component you specify the type of component you wish to fit. For each component you need an estimate before fitting.

You can either do this interactively or automatically. To make an interactive estimate, select the component from the TAB-widget. Then press the 'Inter' button and follow the instructions in the message widget at the bottom of the GUI.

To make an automatic estimate, just press the 'Estimate' button at the bottom of the GUI. It will make an estimate for as many components as you have specified (i.e. you don't do it for each component separately).

The estimate is plotted for you. Then you can press the 'Fit profile' button and the fit will be done. The fit and residual will be plotted.

If you don't want to see all of the plots (data, estimate, fit, residual), you can select what you want from the 'File' menu of the menubar at the top of the GUI.

That's it ! If you want to fix parameters of the fit at the estimate value, check the 'Fix' button for the appropriate component parameter. You can transfer the fit solution to the estimate for each parameter by pressing the left-pointing arrow.

The results are presently stored in a *Glish* record. You can copy it to the **defaultclipboard** `tool` by pressing the 'Copy' button at the bottom the GUI. You can then access the results from *Glish* by pasting from the clipboard.

```
- r := dcb.paste()      # Paste from clipboard to record
- print field_names(r)
estimate fit position
- print field_names(r.fit)
elements xunit xabs yunit doppler nmax baseline
```

Presently, the results are held in a *Glish* record with fields 'estimate', 'fit' and 'position' (the structure of this record is **very** likely to change in the future). There are a variety of subfields with obvious names. To access the results of the fit you would look at the `fit.elements` field of the record (above). This record is indexed by the number of components (although *Glish* record indexing is rather perversely named; you see funny field names like *445, *446 - ignore these). What you would do to see the individual components, assuming say, you fitted two components to your spectrum would be

```
- print r.fit.elements[1]
[type=GAUSSIAN, parameters=[0.60423172 1.4136544 0.00022220827] ,
                errors=[0.042635564 7.6873681e-06 1.8104964e-05] ]
- print r.fit.elements[2]
```

```
[type=GAUSSIAN, parameters=[0.13442206 1.4142888 4.6772192e-05] ,
                errors=[0.093013123 1.588148e-05 3.7512798e-05] ]
```

You can also add the result from each fit to an internal store via the 'Store' button at the bottom of the GUI. The internal store of all the fits can be recovered with the function getstore.

Fitting many profiles in a region
This mode is active when you have selected the Region Usage menu to say 'Fit all profiles'.
From the Viewer control buttons (buttons to the left of the display of the image), select one of the region handling buttons (box or polygon). Now go to the image display, and generate the region by dragging out the box or marking the polygon. As always, double click within the region to emit it. This will cause a second GUI to appear. Is is the same GUI as described above for fitting individual profiles.
The spectrum that is plotted, is the spectrum at the bottom-left corner of your region. You use this GUI to indicate how many components you want to fit, what parameters are fixed and if you wish, and to set an initial estimate for the first profile. Thereafter, an automatic (independent) estimate is made for each profile.
When you press the 'Fit region' button, all profiles in the delineated region are automatically fit. Two new output images are created for you. They have the same name as the input image but with `.fit` and `.resid` appended to them. These images hold the evaluated fit and resdidual.

Non-interactve profile selection
Imageprofilefitter also has a mode where the image is not displayed (`showimage=F` in constructor). This means you cannot interactively select the spatial location at which the next profile will be extracted.
What happens then is that the current image (set in the constructor or via function setimage) is used to generate a profile averaged over the entire image (for the non-profile axes). It is used in this way in Dish; each single-dish spectrum is put into a 1-D image and passed to Imageprofilefitter.

Frame conversions
At the moment, when you plot profiles, you will find that spectral frame conversions are disabled because that functionality is not yet implemented in the fitter (although the plotter can handle it). If you really need to fit in a different reference frame, you can do something like the following:

Example

```
- im := image('ngc1313.cube')
- cs := im.coordsys()
- cs.setconversiontype (spectral='TOPO')       # Convert from native frame to TOPO
```

```
- im.setcoordsys(cs)
- cs.done()
- im.done()
- p := imageprofilefitter('ngc1313.cube')
```

and now the plotting and fitting will be done in the TOPOcentric frame.

Future Enhancements
This is the first version of this `tool`. Future improvements will include

- Enable and handle reference frame changes (e.g. LSRK to BARY etc)
- Line shapes other than Gaussian (e.g. Zeeman, Voigt)
- Line-ratio constraints
- Interactive capability when fitting many profiles in the region
- Robust estimates of line parameters
- Zooming in and out of the profile display and setting ranges

imageprofilefitter - Constructor

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

Construct an imageprofilefitter

Synopsis
imageprofilefitter(infile, infile2, sigma, parent, axis, plotter, showimage, widgetset)

Arguments

infile	in	The primary image file name or image tool Allowed:String or Image tool
infile2	in	The secondary image file name or image tool Allowed:String or Image tool Default: Auto generate
sigma	in	Weights image Allowed:String Default: None
parent	in	Parent frame to embed GUI in Allowed:Tk frame agent Default: F
axis	in	Profile pixel axis Allowed:Integer Default: Spectral
plotter	in	The plotter for profile plots Allowed:unset or a pgplotter tool Default: Will create its own
showimage	in	Display the image ? Allowed:T or F Default: T
widgetset	in	The widgetset to use for the GUI Allowed:A widgetserver tool Default: ddlws

Returns
imageprofilefitter tool

Description

This constructor creates the primary GUI. You must specify the primary image via the `infile` argument. The secondary image specified by `infile2` is optional. If you give it, this is the image that is displayed from which you indicate which profiles to fit. If you don't give it, it is generated automatically from the primary image by averaging the data over the profile axis with a $3 - \sigma$ clip applied. You can specify the input images either by their file names or by giving an `image tool`.

You may specify a fitting weights image via argument `sigma`. It must have the same shape as the primary image. If you don't give it, all pixels have the same weight in the fitting process.

The profile axis is specified by the `axis` argument. If you leave it unset, then the spectral axis, if there is one, is selected. Otherwise axis 3 is selected.

You may optionally embed the display in an existing Tk frame (`parent`) if you wish.

By default, the Pgplotter `tool` that is used to display the profiles is created for you. But you may wish to pass in your own plotter (perhaps it's embedded in some other GUI). If so, use the `plotter` argument to pass in a Pgplotter `tool`. Finally, `showimage`, when set to F, instructs the Imageprofilefitter not to display the image. This means you cannot select profiles interactively. What happens then is that the current image (set in the constructor or via function setimage) is used to generate a profile averaged over the entire image (for the non-profile axes).

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube', infile2='locate.image', axis=3)
- pf.done()
```

---

imageprofilefitter.getestimate - Function

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

Recover the estimate

Synopsis
getestimate()

Returns
Glish record

Description

When you are fitting individual profiles, this returns the current estimate into a record. If you are fitting many profiles in a region this function returns an empty record.
Field 'elements' holds the parameters of each component. Thus the length of field 'elements' is the number of components you fit. Its subfields are self-explanatory when you look at it.
Field 'xunit' gives the abcissa unit. Field 'xabs' says whether the x-values are absolute or relative (to reference pixel). Field 'yunit' gives the ordinate unit. Field 'doppler' gives the doppler type if the profile axis is spectral.

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube')
- est := pf.getestimate()
```

---

imageprofilefitter.getfit - Function

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

Recover the fit

Synopsis
getfit()

Returns
Glish record

Description

This returns the current fit into a record.
If you are fitting individual profiles then the returned record has two fields, 'fit' and 'position'. The 'fit' field holds all the fields described in function getestimate. The 'position' field holds the position of the profile in a variety of formats (pixel, world, string formatted etc). It's self explanatory on inspection.
If you are fitting many profiles in a region then the length of this record is the number of profiles you fit in the region. Each field holds subfields 'position' and 'fit'. The 'fit' subfield has all the subfields described in getestimate.

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube')
- fit := pf.getfit()
```

---

imageprofilefitter.getprofiles - Function

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

Get data, estimate and fit as vectors

Synopsis
getprofiles(outfile, ascii, format, dataonly, overwrite, ack)

Arguments

outfile	in	Output file Allowed:String Default: None
ascii	in	Save in ascii file or aips++ table ? Allowed:Boolean Default: F
format	in	Formatting string for ascii table Allowed:String Default: %e
dataonly	in	Only write out data, not estimates and/or fits Allowed:Boolean Default: F
overwrite	in	Overwrite output file if it exists ? Allowed:Boolean Default: T
ack	in	Acknowledge Table creation/writing to logger ? Allowed:Boolean Default: F

Returns
Record or T

Description

This function recovers vectors holding the abcissa, data, estimate, fit, residual (these three are optional) and mask for the interactive fit just done. The abcissa is saved in both absolute pixel and world coordinates. For the latter, it will be in the units (and doppler if relevant) as displayed on the plotter.
If you don't specify an output file name, these vectors are returned in a record to you with field names as above.
If you specify an output file name you can save the data as an aips++ Table (default) or an ascii file. For the latter you can specify the floating point format. For the latter the mask will be stored as 0 (bad) or 1 (good).
If you store as an AIPS++ table, the units are stored as column keywords. If you store as an ascii table, the units are written as the first line. If the abcissa

units are consistent with velocity, then the doppler type is written out as well. So you will see on the first line, 'xunit yunit' or 'xunit doppler yunit'. The second line gives the column headers.

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube', infile2='locate.image', axis=3)
-                                      # User does a fit
- r := pf.getprofiles();               # Recover into record
- field_names(r);
abcissa data estimate fit residual mask
-
- pf.getprofiles(outfile='junk.tbl')  # Save as aips++ table
- t := table('junk.tbl')               # Open
- t.browse();                          # Browse
```

---

imageprofilefitter.getstore - Function

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

Recover all fits added to internal store

Synopsis
getstore()

Returns
Glish record

Description

This returns all of the fits that you have added to the internal store (by pressing the 'Store' button) into a record.
There is one (numerically labelled) field in the record for each press of the 'Store' button.
Each of these records has further fields 'estimate', 'fit', and 'position' holding the obvious things.

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube')

# do some fits

- s := pf.getstore();
- print 'Number of components = ', length(s)
Number of components = 10
-
- field_names(s[5])
estimate fit position
```

---

imageprofilefitter.done - Function

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

Destroy this imageprofilefitter tool

Synopsis
done()

Returns
Bool or fail

Description
This function destroys the Imageprofilefitter `tool`.

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube')
- pf.done()
- pf
F
```

---

imageprofilefitter.gui - Function

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

Start the GUI interface

Synopsis
gui()

Returns
Bool or fail

Description
If you dismiss the GUI (with the dismiss button), then you can recover it with this function. The state of the Imageprofilefitter and the GUI is preserved. If the GUI is already present, then it does nothing.

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube')
- pf.gui()
```

---

imageprofilefitter.setimage - Function

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

Set a new image

Synopsis
setimage(infile, infile2, sigma)

Arguments

infile	in	The primary image file name or image tool Allowed:String or Image tool
infile2	in	The secondary image file name or image tool Allowed:String or Image tool Default: Auto generate
sigma	in	Weights image Allowed:String Default: None

Returns
Bool or fail

Description
You can set a new image (primary and secondary) with this function. Doing so will cause the displays to be updated.
You may specify a fitting weights image via argument `sigma`. It must have the same shape as the primary image. If you don't give it, all pixels have the same weight in the fitting process.

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube')
- pf.setimage(infile='sgra.cube')
```

---

imageprofilefitter.type - Function

Package	**general**
Module	**images**
Tool	**imageprofilefitter**

What type is this tool ?

Synopsis
type()

Returns
String or fail

Description
Returns the string "imageprofilefitter".

Example

```
- pf := imageprofilefitter(infile='ngc1313.cube')
- pf.type()
imageprofilefitter
```

1.2.17 imagepol - Tool

Package general
Module images

Polarimetric analysis of images
include "imagepol.g"

Constructors

imagepol	Create an Imagepol tool from an AIPS++ image file
imagepoltestimage	Create an Imagepol tool and a test image file

Functions

complexfraclinpol	Complex fractional linear polarization
complexlinpol	Complex linear polarization
depolratio	Linear depolarization ratio
done	Destroy this Imagepol tool
fourierrotationmeasure	Find Rotation Measure (Fourier approach)
fraclinpol	Fractional linear polarization
fractotpol	Fractional total polarization
id	Return the fundamental identifier of this tool
linpolint	Linearly polarized intensity
linpolposang	Linearly polarized position angle
makecomplex	Make a Complex image
pol	Polarized quantities
rotationmeasure	Find Rotation Measure (traditional approach)
sigma	Find best guess at thermal noise
sigmadepolratio	Error in linear depolarization ratio
sigmafraclinpol	Error in fractional linear polarization
sigmafractotpol	Error in fractional total polarization

sigmalinpolint	Error in linearly polarized intensity
sigmalinpolposang	Error in linearly polarized position angle
sigmastokes	Find standard deviation of specified Stokes data
sigmastokesi	Find standard deviation of Stokes I data
sigmastokesq	Find standard deviation of Stokes Q data
sigmastokesu	Find standard deviation of Stokes U data
sigmastokesv	Find standard deviation of Stokes V data
sigmatotpolint	Error in total polarized intensity
stokes	Stokes
stokesi	Stokes I
stokesq	Stokes Q
stokesu	Stokes U
stokesv	Stokes V
summary	Summarise Imagepol tool
totpolint	Total polarized intensity

Summary

An Imagepol `tool` provides specialized polarimetric analysis of images. Some of these things could be done with the Lattice Expression Language (LEL; see note223) but are more conveniently offered separately.

General

You construct an Imagepol tool from an image (AIPS++, FITS, and Miriad formats are supported) with a Stokes coordinate axis. What you can do with your Imagepol `tool` depends on exactly which Stokes parameters you have in the image. You must have some combination of Stokes I, Q, U and V on the Stokes axis. These refer to total intensity, two components of linear polarization, and circular polarization, respectively. Therefore, if you ask for linear polarization and the image only has Stokes I and V, you will get an error.

The Imagepol `tool` functions generally return, by default, an Image `tool` as their output. In most cases, this is a "virtual" image. There are a range of different sorts of "virtual" images in AIPS++ (see Image). But the Imagepol `tool functions` generally return reference Image `tools`. That is, these reference different pieces of the original image that was used to construct the Imagepol `tool`, either directly, or as mathematical expressions (e.g. the polarized intensity). If you delete the construction image, you render your Imagepol `tool` and its outputs useless. If you wish, rather than return a virtual image `tool`, you can fill in the `outfile` argument of most Imagepol `tool` functions and write the image out to disk, associating the Image `tool` with the disk file.

In some of the functions, the standard deviation of the thermal noise is needed. This is for debiasing polarized intensity images or working out statistical error images. By default it is worked out for you from the construction image with outliers from the mean discarded. Since it is the thermal noise it is trying to find, it is worked out from the V, Q & U, and finally I data in that order of precedence. This is because Stokes V is much less likely to contain source signal than Stokes I. You can supply the noise level if you know it better. For example, for small images or images with few signal-free pixels, the theoretical estimate may be better.

Analysis and Display

Traditionally, when generating secondary and tertiary images (e.g. position angle, fractional polarization, rotation measure etc), one masks the putput image according to some statistical test. For example, if the error in the output image is greater than some value, or the errors in the input images are greater than some value. Imagepol `tools` do not offer this kind of masking. It does provide you with the error images for the derived images. By using LEL when you analyze your images, you can mask the images however you want when you use them. That is, we defer the error interpretation as long as possible. Here is an example.

Example

```
include 'imagepol.g'
ip := imagepol('stokes.image')       # Construct Imagepol tool from image
lpa := ip.linpolposang()             # Linearly polarized position angle image
lpaerr := ip.sigmalinpolposang()     # Error in linearly polarized position angle image
lpa.view(mask='$lpaerr<5')           # Display when p.a. error < 5 degrees
```

Display is handled via the Viewer `tool`. It can display and overlay combinations of raster, contour and vector representations of your data. Please see the view function of the Image `tool` and the document Getting Results) for high-level information on how to display images in general.
It is common to display linear polarization data via vectors where the position angle of the vector is the position angle of the linear polarized radiation, and the amplitude of the vector is proportional to either the total polarized intensity or fractional polarized intensity.
The data source of a vector display is either a Complex or a Float image. If it is a Complex image (e.g. the complex linear polarization $Q + iU$) then both the amplitude and the phase (position angle) are available. If it is just a Float image, then it is assumed to be the position angle and an amplitude of unity will be provided. The angular units are given by image brightness units which you can set with function setbrightnessunits. If the units are not recognized as angular, degrees are assumed.
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.

The Imagepol `tool` can create Complex disk images for you via functions complexlinpol (complex linear polarization), complexfraclinpol (complex fractional linear polarization) and makecomplex (takes amp/phase or real/imag). As well as these Complex images, you can also make Float images of the linearly polarized intensity, linearly polarized position angle, and the fractional linearly polarized intensity (see below).
Now, the Image `tool` cannot yet deal with Complex images (it will in the future). This means that you cannot currently do

```
include 'imagepol.g'
p := imagepol('iquv.im')  # Make Imagepol tool
p.complexlinpol('clp')    # Make complex image of linear polarization disk file
clp := image('clp')       # Error
```

which is a bit annoying presently. However, the Viewer `tool` is able to read Complex images so that you are able to display them ok.

```
include 'imagepol.g'
include 'viewer.g'
p := imagepol('iquv.im')  # Make Imagepol tool
p.complexlinpol('clp')    # Make complex image of linear polarization disk file
dv.gui()                  # Start data manager to give access to Complex image
```

If you wanted to make a vector map display you would select the appropriate image in the Viewer's data manager GUI, click 'Vector Map' on the right hand side and it would appear in the display. Note that the Viewer's Vector map display capability also offers you amplitude noise debiasing and the On-The-Fly mask.

Overview of Imagepol `tool` functions

- Access to the different Stokes subimages is via functions stokes, stokesi, stokesq, stokesu, and stokesv.
- Create the standard secondary and tertiary polarization images with
  - complexfraclinpol - complex fractional linear polarization
  - complexlinpol - complex linear polarization
  - makecomplex - make complex image from amp/phase or real/imag
  - pol - polarized quantities as specified
  - linpolint - linearly polarized intensity
  - linpolposang - linearly polarized position angle
  - totpolint - total polarized intensity
  - fraclinpol - fractional linear polarization

  - fractotpol - fractional total polarization
  - depolratio - linear depolarization ratio

- Create the standard secondary and tertiary polarization error images (simple propagation of Gaussian errors) with

  - sigma - best guess at thermal noise
  - sigmastokes - specified Stokes parameter
  - sigmastokesi - Stokes I
  - sigmastokesq - Stokes Q
  - sigmastokesu - Stokes U
  - sigmastokesv - Stokes V
  - sigmalinpolint - linearly polarized intensity
  - sigmalinpolposang - linearly polarized position angle
  - sigmatotpolint - total polarized intensity
  - sigmafraclinpol - fractional linear polarization
  - sigmafractotpol - fractional total polarization
  - sigmadepolratio - linear depolarization ratio

- You can compute Rotation Measure images via either the traditional techniques involving a number of different frequencies (regularly or irregularly spaced) with function

  rotationmeasure or via a new Fourier Technique for many regularly-spaced frequencies with function fourierrotationmeasure.

imagepol - Constructor

Package	**general**
Module	**images**
Tool	**imagepol**

Create an Imagepol tool from an AIPS++ image file

Synopsis
imagepol(infile)

Arguments

infile	in	Input image (file name or tool) Allowed:String or Image tool

Returns
Imagepol tool or fail

Description

The constructor takes an image as its input. This can be specified as either the name of a disk file, or you can give a pre-existing Image `tool`. If you give a disk file name, the disk image file may be in AIPS++, FITS or Miriad format. The input image must have a Stokes axis. The exact collection of Stokes that the image has, determines what the Imagepol `tool` can compute. Stokes I, Q, U, and V refer to total intensity, two components of linear polarization, and circular polatization, respectively. Therefore, if you ask for linear polarization and the image only has Stokes I and V, you will get an error.
The input image may contain data at many frequencies. For example, the image may be a 4D image with axes RA, DEC, Stokes and Frequency (order not important) where the Frequency axis is regularly sampled. However, the image may also contain many frequencies at irregular intervals. Such an image may be created with the Image `tool` constructor called imageconcat. It enables you to concatenate images along an axis, and it allows irregular coordinate values along that axis.

Example

```
- im := image('$AD/myimage')
- p1 := imagepol(im)
- p2 := imagepol('$AD/myimage')
```

We create two Imagepol `tools`, one from an Image `tool` and one directly from the disk file.

Example

```
- im := imageconcat(infiles="im.f1 im.f2 im.f3 im.f4 im.f5", axis=4)
- p := imagepol(im)
- p.rotationmeasure(rm='rm', rmerr='rmerr', rmmax=500, maxpaerr=10)
```

Say we have 5 images, each with axes RA, DEC, Stokes, and Frequency in that order. We make an Image `tool` which concatenates these images along the frequency axis - you have ordered them in increasing or decreasing frequency order. Note that the Image `tool` is virtual - it is not written to an output file. This just means that the data are read from the input images as needed. The Imagepol `tool` is then made from this virtual Image `tool`. We then compute the Rotation Measure and Rotation Measure error images with the traditional method and write them out to disk.

---

imagepoltestimage - Constructor

Package	**general**
Module	**images**
Tool	**imagepol**

Create an Imagepol tool and a test image file

Synopsis

imagepoltestimage(outfile, rm, pa0, sigma, nx, ny, nf, f0, bw)

Arguments

outfile	in	Output image file name Allowed:String Default: imagepol.iquv
rm	in	Rotation Measure (rad/m/m) Allowed:Vector of float Default: Auto no-ambiguity determine
pa0	in	Position angle (degrees) at zero wavelength Allowed:Float Default: 0
sigma	in	Fractional noise level Allowed:Float Default: 0.001
nx	in	Shape of image in x direction Allowed:Integer Default: 32
ny	in	Shape of image in y direction Allowed:Integer Default: 32
nf	in	Shape of image in frequency direction Allowed:Integer Default: 32
f0	in	Reference frequency (Hz) Allowed:Float Default: 1.4e9
bw	in	Bandwidth (Hz) Allowed:Float Default: 128e6

Returns
Imagepol `tool` or fail

Description

The constructor can be used to generate a test image and then create an Imagepol `tool` from it.
The test image is 4-dimensional (RA, DEC, Stokes and Frequency). The Stokes axis holds I,Q,U and V. The source is just a constant I (if you don't add noise all spatial pixels will be identical) and V. Q and U vary with frequency according to the specified Rotation Measure components (no attempt to handle bandwidth smearing within channels is made). The actual values of I,Q,U, and V are chosen arbitrarily otherwise (could be added as arguments if desired).
You can use this image, in particular, to explore the Rotation Measure algorithms in functions rotationmeasure and fourierrotationmeasure.
If you don't specify the Rotation Measure, then it is chosen for you so that there is no position angle ambiguity between adjacent channels (the value will be sent to the Logger).
The noise added to the image is specified as a fraction of the total intensity (constant). Gaussian noise with a standard deviation of `sigma *` I_max is then added to the image.

Example

```
- p1 := imagepoltestimage(rm=200)
- p1.rotationmeasure(rm='rm.out', rmmax=250)
- rm := image('rm.out')
- rm.view()
```

---

imagepol.complexlinpol - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Complex linear polarization

Synopsis
complexlinpol(outfile)

Arguments

outfile	in	Output image file name Allowed:String

Returns
T or fail

Description

This function (short-hand name `clp`) produces the complex linear polarization; $Q + iU$ and writes it to a disk image file.
The Image `tool` cannot yet handle Complex images. You must therefore write the Complex image to disk. The Viewer can display Complex images. Also the Table tool can access Complex images.

Example

```
- include 'imagepol.g'
- include 'table.g'
- p := imagepol('stokes.cube')
- p.complexlinpol('cplx')
- t := table('cplx')
- t.browse()
```

---

imagepol.complexfraclinpol - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Complex fractional linear polarization

Synopsis
complexfraclinpol(outfile)

Arguments

outfile	in	Output image file name Allowed:String

Returns
T or fail

Description

This function (short-hand name `clp`) produces the complex fractional linear polarization; $(Q + iU)/I$ and writes it to a disk image file.
The Image `tool` cannot yet handle Complex images. You must therefore write the Complex image to disk. The Viewer can display Complex images. Also the Table tool can access Complex images.

Example

```
- include 'imagepol.g'
- include 'table.g'
- p := imagepol('stokes.cube')
- p.complexfraclinpol('cplx')
- t := table('cplx')
- t.browse()
```

---

imagepol.depolratio - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Linear depolarization ratio

Synopsis
depolratio(infile, debias, clip, sigma, outfile)

Arguments

infile	in	Other image Allowed:String or Image tool
debias	in	Debias the linearly polarized intensity ? Allowed:T or F Default: F
clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description

This function (short-hand name `dr`) returns the linear depolarization ratio computed from two frequencies; this is the ratio of the fractional linear polarization at the two frequencies. Generally this is done when you have generated two images, each at a different frequency (continuum work). Thus if the fractional linear polarization images are $m_{\nu 1}$ and $m_{\nu 2}$ then the depolarization ratio is $m_{\nu 1}/m_{\nu 2}$.
This function operates with two images; the first (at frequency $\nu 1$) is that with which you constructed your Imagepol `tool`. The second (at frequency $\nu 2$) is supplied via the argument `infile`, which can be an Image `tool`, or a String holding the name of the `image file`.
In generating the depolarization ratio, you may optionally debias the linearly polarized intensity. This requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.
You can get the depolarization ratio error image with function sigmadepolratio.

Example

```
- im1 := image('stokes.4800')
- im2 := image('stokes.8300')
- p := imagepol(im1)
```

```
- dpr := p.depolratio(im2);          # m_4800 / m_8300
- edpr := p.sigmadepolratio(im2);
```

---

imagepol.done - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Destroy this Imagepol tool

Synopsis
done()

Returns
T or fail

Description

If you no longer need to use an Imagepol `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`. Note that the underlying `image file` with which the `tool` was constructed is not deleted. In addition, any images that you created with the Imagepol `tool functions` will also still be viable.

Example

```
- p := imagepol('stokes.cube')
- p.done()
- p
F
```

---

imagepol.fourierrotationmeasure - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Find Rotation Measure (Fourier approach)

Synopsis
fourierrotationmeasure(complex, amp, pa, real, imag, zerolag0)

Arguments

complex	in	Output complex linear polarization image file name Allowed:String Default: Unset
amp	in	Output linear polarization amplitude image file name Allowed:String Default: Unset
pa	in	Output linear polarization position angle (degrees) image file name Allowed:String Default: Unset
real	in	Output linear polarization real image file name Allowed:String Default: Unset
imag	in	Output linear polarization imaginary angle image file name Allowed:String Default: Unset
zerolag0	in	Force zero lag to 0 ? Allowed:T or F Default: F

Returns
T or fail

Description

This function (short-hand name `frm`) will only work if you constructed the Imagepol `tool` from an image containing Stokes Q and U, and a regular frequency axis. It Fourier transforms the complex linear polarization (Q+iU) over the spectral axis to the rotation measure axis. Thus, if your input image contained RA/DEC/Stokes/Frequency, the output image would be RA/DEC/RotationMeasure. The Rotation Measure axis has as many pixels as the spectral axis.
This method enables you to see the polarization as a function of Rotation Meausure. Its main use is when searching for large RMs. See Killeen, Fluke,

Zhao and Ekers (1999, preprint) for a description of this method (or http://www.atnf.csiro.au/~nkilleen/rm.ps) and its advantages over the traditional method (rotationmeasure) of extracting the Rotation Measure.
Although you can write out the complex polarization image with the argument `complex`, you can't do much with it because Image `tools` cannot yet be constructed from complex images. Hence you can also write out the complex linear polarization image in any or all of the other forms.
The argument `zerolag0` allows you to force the zero lag (or central bin) of the Rotation Measure spectrum to zero (effectively by subtracting the mean of Q and U from the Q and U images). This may avoid Gibbs phenomena from any strong low Rotation Measure signal which would normally fall into the central bin.

Example

```
- p := imagepoltestimage(outfile='iquv.im', rm=[5.0e5,1e6], nx=8, ny=8, nf=512,
                         f0=1.4e9, bw=8e6)
- p.frm(amp='amp')
- amp := image('amp')
- amp.view()                    # And reorder to put RM along X-axis
```

In this example we give two RM components and recover them.

---

imagepol.fraclinpol - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Fractional linear polarization

Synopsis
fraclinpol(debias, clip, sigma, outfile)

Arguments

debias	in	Debias the linearly polarized intensity ? Allowed:T or F Default: F
clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description
This function (short-hand name `flp`) returns the fractional linear polarization; $\sqrt{(Q^2 + U^2)}/I$.
You may optionally debias the polarized intensity. This requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- flp := p.fraclinpol()
```

---

imagepol.fractotpol - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Fractional total polarization

Synopsis
fractotpol(debias, clip, sigma, outfile)

Arguments

debias	in	Debias the total polarized intensity ? Allowed:T or F Default: F
clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description
This function (short-hand name `ftp`) returns the fractional linear polarization; $\sqrt{(Q^2 + U^2 + V^2)}/I$.
You may optionally debias the polarized intensity. This requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.
If your image contains only Q and U, or only V, then just those components contribute to the total polarized intensity.

Example

```
- p := imagepol('stokes.cube')
- ftp := p.fractotpol()
```

---

imagepol.id - Function

Package	**general**
Module	**images**
Tool	**imagepol**

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

```
- p := imagepol('stokes.cube')
- p.id()
[sequence=15, pid=20665, time=941768782, host=elara, agentid=3]
```

---

imagepol.linpolint - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Linearly polarized intensity

Synopsis
linpolint(debias, clip, sigma, outfile)

Arguments

debias	in	Debias the linearly polarized intensity ? Allowed:T or F Default: F
clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description
This function (short-hand name `lpi`) returns the linearly polarized intensity; $\sqrt{(Q^2 + U^2)}$.
You may optionally debias the polarized intensity. This requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- lpi := p.linpolint()
```

---

imagepol.linpolposang - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Linearly polarized position angle

Synopsis
linpolposang()

Returns
Image tool or fail

Description

This function (short-hand name `lppa`) returns the linearly polarized position angle image ($0.5\tan^{-1}(U/Q)$) in degrees.

Example

```
- p := imagepol('stokes.cube')
- lppa := p.linpolposang()
```

---

imagepol.makecomplex - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Make a Complex image

Synopsis
makecomplex(complex, real, imag, amp, phase)

Arguments

complex	in	Output complex image file name Allowed:String
real	in	Input real image file name Allowed:String Default: Unset
imag	in	Input imaginary image file name Allowed:String Default: Unset
amp	in	Input amplitude image file name Allowed:String Default: Unset
phase	in	Input phase image file name Allowed:String Default: Unset

Returns
T or fail

Description

This function generates a Complex `image file` from either a real and imaginary, or an amplitude and phase pair of images. If you give a linear position angle image for the phase, it will be multipled by two before the real and imaginary parts are formed.

Example

```
- p := imagepol('iquv')
- p.complexlinpol('qu.cplx1')
-
- q := p.stokesq()
- u := p.stokesu()
- q2 := q.subimage('q')
- u2 := u.subimage('u')
- p.makecomplex('qu.cplx2', real='q', image='u')
```

In this example we make two complex linear polarization images which should be identical.

---

imagepol.pol - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Polarized quantities

Synopsis
pol(which, debias, clip, sigma, outfile)

Arguments

which	in	Specify operation Allowed:String: One of 'lpi', 'tpi', 'lppa', 'flp', 'ftp' (case insensitive)
debias	in	Debias the polarized intensity ? Allowed:T or F Default: F
clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description

This function just packages the other specific polarization functions into one where you specify an operation with the argument `which` (can be useful for scripts). This argument can take the values:

- 'lpi' - linearly polarized intensity (function linpolint)
- 'tpi' - total polarized intensity (function totpolint)
- 'lppa' linearly polarized position angle (function linpolposang)
- 'flp' - fractional linear polarization (function fraclinpol)
- 'ftp' - fractional total polarized intensity (function fractotpol)

Example

```
- p := imagepol('stokes.cube')
- lpi := p.pol('lpi')
```

---

imagepol.rotationmeasure - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Find Rotation Measure (traditional approach)

Synopsis
rotationmeasure(rm, rmerr, pa0, pa0err, nturns, chisq, sigma, rmfg, rmmax, maxpaerr, plotter, nx, ny)

Arguments

rm	in	Output Rotation Measure image file name Allowed:String Default: Unset
rmerr	in	Output Rotation Measure error image file name Allowed:String Default: Unset
pa0	in	Output position angle (degrees) at zero wavelength image file name Allowed:String Default: Unset
pa0err	in	Output position angle (degrees) at zero wavelength error image file name Allowed:String Default: Unset
nturns	in	Output number of turns image file name Allowed:String Default: Unset
chisq	in	Output reduced chi squared image file name Allowed:String Default: Unset
sigma	in	Estimate of the thermal noise Allowed:Float Default: Auto
rmfg	in	Foreground Rotation Measure (rad/m/m) to subtract Allowed:Float Default: 0
rmmax	in	Maximum Rotation Measure (rad/m/m) to solve for Allowed:Float Default: 0
maxpaerr	in	Maximum input position angle error (degrees) to allow Allowed:Float Default: None
plotter	in	Name of plotter Allowed:String Default: None
nx	in	Number of plots in x direction Allowed:Integer Default: 5
ny	in	Number of plots in y direction Allowed:Integer Default: 5

Returns
T or fail

Description

This function (short-hand name `rm`) generates the Rotation Measure image from a collection of different frequencies. It will only work if the Imagepol `tool` was constructed from an image containing Stokes Q and U, and a frequency axis (regular or irregular) with at least 2 pixels. It will work out the position angle images for you.

See also the fourierrotationmeasure function for a new Fourier-based approach.

Rotation Measure algorithms that work robustly are not common. The main problem is in trying to account for the $n-\pi$ ambiguity (see Leahy et al, Astronomy & Astrophysics, 156, 234 or Killeen et al; http://www.atnf.csiro.au/~nkilleen/rm.ps).

The algorithm that this function uses is that of Leahy et al. Please refer to their paper for details (their Appendix A.1). But as in all these algorithms, the basic process is that for each spatial pixel, a vector of position angles (i.e. at the different frequencies) is fit to determine the Rotation Measure and the position angle at zero wavelength (and their errors). You can write out an image containing the number of $n--\pi$ turns that were added to the data at each spatial pixel and for which the best fit was found. You can also write out the reduced chi-squared image for the fits.

Note that as yet no assessment of curvature (i.e. deviation from the simple linear position angle - λ^2 functional form) is made.

You can write out any or all of the output images.

The argument `sigma` gives the themal noise in Stokes Q and U. By default it is worked out for you from the image data. But if it proves to be inaccurate (maybe not many signal-free pixels), then you can input it here. This is used for working out the error in the position angles (propagation of Gaussian errors).

The argument `maxpaerr` specifies the maximum allowable error in the position angle that is acceptable. The default is an infinite value. From the standard propagation of errors, the error in the linearly polarized position angle is determined from the Stokes Q and U images (at each spatial pixel for each frequency). At each spatial pixel we do a fit to the position angle vector (i.e. at the different frequencies) to determine the Rotation Measure. If the position angle error for any pixel in the vector exceeds the specified value, it is dropped from the fit. The process generates an error for the fit and this is used to compute the errors in the output images.

Note that `maxpaerr` is *not* used to specify that any pixel for which the output position angle error exceeds this value should be masked out.

The argument `rmfg` is used to specify a foreground RM value. For example, you may know the mean RM in some direction out of the Galaxy, then including this can aid the algorithm (reduces ambiguity).

The argument `rmmax` specifies the maximum absolute RM that should be solved for. This quite an important parameter. If you leave it at the default, zero, basically, no ambiguity handling will be invoked. So you must have some

idea of what you are looking for - this is the basic problem with Rotation Measure algorithms.
Plotting can be done with a PGPLOT device via argument `plotter`. The syntax is `plotter=name/type`. For example `plotter='plot1.ps/ps'` (disk postscript file) or `plotter='1/xs'` (X-windows device) or `plotter='plot/glish'` (*Glish* PGplotter). The plots show the used position angle data (after ambiguity adjustments), position angle errors, and fit for each profile. The pixel location of each profile and reduced chi-square of the fit are written in the title.

Example

```
- im := imageconcat(infiles="im.f1 im.f2 im.f3 im.f4 im.f5", axis=4)
- p := imagepol(im)
- ok := p.rm(rm='rm', rmerr='rmerr', rmmax=800, maxpaerr=10)
```

Say we have 5 images, each with axes RA, DEC, Stokes, and Frequency in that order. We make an Image `tool` which concatenates these images along the frequency axis - you have ordered them in increasing or decreasing frequency order. Note that the Image `tool` is virtual - it is not written to an output file. This just means that the data are read from the input images as needed. The Imagepol `tool` is then made from this virtual Image `tool`. We then compute the Rotation Measure and Rotation Measure error images with the traditional method and write them out to disk.

---

imagepol.sigma - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Find best guess at thermal noise

Synopsis

sigma(clip)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10

Returns
Float or fail

Description

This function returns the standard deviation from V, Q&U or I in that order of precedence. It is attempting to give you the best estimate of the thermal noise it can from the data. Outliers from the mean are clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigma := p.sigma()
```

---

imagepol.sigmadepolratio - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Error in linear depolarization ratio

Synopsis
sigmadepolratio(infile, debias, clip, sigma, outfile)

Arguments

infile	in	Other image Allowed:String or Image tool
debias	in	Debias the linearly polarized intensity ? Allowed:T or F Default: F
clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description

This function (short-hand name `edr`) returns the error in the linear depolarization ratio computed from two frequencies; this is the ratio of the fractional linear polarization at the two frequencies. Generally this is done when you have generated two images, each at a different frequency (continuum work). Thus if the fractional linear polarzation images are $m1$ and $m2$ then the depolarization ratio is $m1/m2$.
This function operates with two images; the first is that with which you constructed your Imagepol `tool`. The second is supplied via the argument `infile`, which can be an Image `tool`, or a String holding the name of the `image file`.
In generating the depolarization ratio, and hence its error, you may optionally debias the linearly polarized intensity. This requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.
You can get the depolarization ratio image with function depolratio.

Example

```
- im1 := image('stokes.4800')
- im2 := image('stokes.8300')
- p := imagepol(im1)
- dpr := p.depolratio(im2);
- edpr := p.sigmadepolratio(im2);
```

imagepol.sigmafraclinpol - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Error in fractional linear polarization

Synopsis

sigmafraclinpol(clip, sigma, outfile)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns

Image tool or fail

Description

This function (short-hand name `sflp`) returns the error (standard deviation) of the fractional linear polarization. This result comes from standard propagation of errors. The result is an Image tool as the error is signal-to-noise ratio dependent.

This function requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigflp := p.sigmafraclinpol()
```

---

imagepol.sigmafractotpol - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Error in fractional total polarization

Synopsis
sigmafractotpol(clip, sigma, outfile)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description

This function (short-hand name `sftp`) returns the error (standard deviation) of the fractional total polarization. This result comes from standard propagation of errors. The result is an Image tool as the error is signal-to-noise ratio dependent.
This function requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigftp := p.sigmafractotpol()
```

---

imagepol.sigmalinpolint - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Error in linearly polarized intensity

Synopsis
sigmalinpolint(clip, sigma, outfile)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Float or fail

Description

This function (short-hand name `slpi`) returns the error (standard deviation) of the linearly polarized intensity; $\sqrt{(Q^2 + U^2)}$. This result comes from standard propagation of statistical errors. The result is a float as the error is not signal-to-noise ratio dependent
This function requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- siglpi := p.sigmalinpolint()
```

---

imagepol.sigmalinpolposang - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Error in linearly polarized position angle

Synopsis
sigmalinpolposang(clip, sigma, outfile)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description

This function (short-hand name `slppa`) returns the error (standard deviation) of the linearly polarized position angle $(0.5\tan^{-1}(U/Q)\sqrt{(Q^2+U^2)})$ in degrees. This result comes from standard propagation of errors. The result is an Image tool as the error is signal-to-noise ratio dependent.
This function requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- siglppa := p.sigmalinpolposang()
```

---

imagepol.sigmastokes - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Find standard deviation of specified Stokes data

Synopsis
sigmastokes(which, clip)

Arguments

which	in	Specify Stokes Allowed:String: One of 'I', 'Q', 'U', 'V' (case insensitive)
clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10

Returns
Float or fail

Description

This function (short-hand name `ss`) returns the standard deviation of the noise for the specified Stokes. Outliers from the mean are clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigq := p.sigmastokes('q', 10.0)
```

---

imagepol.sigmastokesi - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Find standard deviation of Stokes I data

Synopsis
sigmastokesi(clip)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10

Returns
Float or fail

Description

This function (short-hand name `ssi`) returns the standard deviation of the noise for the Stokes I data. Outliers from the mean are clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigi := p.sigmastokesi(10.0)
```

---

imagepol.sigmastokesq - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Find standard deviation of Stokes Q data

Synopsis
sigmastokesq(clip)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10

Returns
Float or fail

Description

This function (short-hand name `ssq`) returns the standard deviation of the noise for the Stokes Q data. Outliers from the mean are clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigq := p.sigmastokesq(10.0)
```

---

imagepol.sigmastokesu - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Find standard deviation of Stokes U data

Synopsis
sigmastokesu(clip)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10

Returns
Float or fail

Description

This function (short-hand name `ssu`) returns the standard deviation of the noise for the Stokes U data. Outliers from the mean are clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigu := p.sigmastokesu(10.0)
```

---

imagepol.sigmastokesv - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Find standard deviation of Stokes V data

Synopsis
sigmastokesv(clip)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10

Returns
Float or fail

Description

This function (short-hand name `ssv`) returns the standard deviation of the noise for the Stokes V data. Outliers from the mean are clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigv := p.sigmastokesv(10.0)
```

---

imagepol.sigmatotpolint - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Error in total polarized intensity

Synopsis

sigmatotpolint(clip, sigma)

Arguments

clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noise Allowed:Float Default: Auto determined

Returns

Float or fail

Description

This function (short-hand name `stpi`) returns the error (standard deviation) of the total polarized intensity; $\sqrt{(Q^2 + U^2 + V^2)}$. This result comes from

standard propagation of statistical errors. The result is a float as the error is not signal-to-noise ratio dependent
This function requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- sigtpi := p.sigmatotpolint()
```

---

imagepol.stokes - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Stokes

Synopsis

stokes(which, outfile)

Arguments

which	in	Specify Stokes Allowed:String: One of 'I', 'Q', 'U', 'V' (case insensitive)
outfile	in	Output image file name Allowed:String Default: Unset

Returns

Image tool or fail

Description

This function returns an image containing the specified Stokes only. This interface can be useful for scripts.

Example

```
- p := imagepol('stokes.cube')
- q := p.stokes('q')
```

---

imagepol.stokesi - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Stokes I

Synopsis
stokesi()

Returns
Image tool or fail

Description
This function returns an image containing Stokes I only.

Example

```
- p := imagepol('stokes.cube')
- i := p.stokesi()
```

---

imagepol.stokesq - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Stokes Q

Synopsis
stokesq()

Returns
Image tool or fail

Description
This function returns an image containing Stokes Q only.

Example

```
- p := imagepol('stokes.cube')
- q := p.stokesq()
```

---

imagepol.stokesu - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Stokes U

Synopsis
stokesu()

Returns
Image tool or fail

Description
This function returns an image containing Stokes U only.

Example

```
- p := imagepol('stokes.cube')
- u := p.stokesu()
```

---

imagepol.stokesv - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Stokes V

Synopsis
stokesv()

Returns
Image tool or fail

Description
This function returns an image containing Stokes V only.

Example

```
- p := imagepol('stokes.cube')
- v := p.stokesv()
```

---

imagepol.summary - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Summarise Imagepol tool

Synopsis
summary()

Returns
Image tool or fail

Description

This function just lists a summary of the Imagepol `tool` to the logger. Currently it just summarizes the image with which it was constructed.

Example

```
- p := imagepol('stokes.cube')
- p.summary()
Image name       : Concatenation :
Telescope        : ATCA
Observer         : Fluke
Date observation : 1998/05/18/09:35:54
Image mask       : None
Image units      : Jy/beam

Direction system : J2000
Frequency system : LSR
Velocity  system : RADIO

Name             Proj Shape Tile   Coord value at pixel    Coord incr Units
---------------------------------------------------------------------------
Right Ascension   SIN   201   67  11:40:48.400   101.00 -2.000000e+00 arcsec
Declination       SIN   201   67 -26.29.30.000   101.00  2.000000e+00 arcsec
Frequency               13    7  1.432000e+09     1.00 -8.000000e+06 Hz
Velocity                                 Fail     1.00          Fail km/s
Stokes                   4    1          IQUV
```

---

imagepol.totpolint - Function

Package	**general**
Module	**images**
Tool	**imagepol**

Total polarized intensity

Synopsis

totpolint(debias, clip, sigma, outfile)

Arguments

debias	in	Debias the total polarized intensity ? Allowed:T or F Default: F
clip	in	Clip level for auto-sigma determination Allowed:Float Default: 10
sigma	in	Standard deviation of thermal noised Allowed:Float Default: Auto determined
outfile	in	Output image file name Allowed:String Default: Unset

Returns
Image tool or fail

Description
This function (short-hand name `tpi`) returns the total polarized intensity; $\sqrt{(Q^2+U^2+V^2)}$. If your image contains only Q and U, or only V, then just those components contribute to the total polarized intensity.
You may optionally debias the polarized intensity. This requires the standard deviation of the thermal noise. You can either supply it if you know it, or it will be worked out for you with outliers from the mean clipped at the specified level.

Example

```
- p := imagepol('stokes.cube')
- tpi := p.totpolint()
```

1.2.18 is_imagepol - Function

Package general
Module images

Is this variable an imagepol tool ?

Synopsis
is_imagepol(thing)

Arguments

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

Returns
T or F

Description

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

Example

```
- p:=imagepol('myimage')
- is_imagepol(p)
T
- p2 := [10,20]
- is_imagepol(p2)
F
```

1.2.19 imagepoltest - Function

Package general
Module images

Run test suite for Imagepol tool

Synopsis
imagepoltest(which)

Arguments

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

Returns
T or Fail

Description

Self-test of the Imagepol `tool`. This function includes forced errors. As long as the function finally returns with a `T`, it has succeeded (regardless of what error messages you might see).
The function invokes many tests. The first one is a general test of everything at a basic level. The succeeding tests work on individual areas of the image module.
Since the number of tests keeps growing, we don't tell you how many there are here or exactly what they do ! If you give too large a value, nothing will happen. `which` can be an integer or a vector of integers.

Example

```
- imagepoltest()
```

Runs all tests.

Example

```
- imagepoltest(2)
```

Runs test 2.

1.2.20 imageprofilesupport - Tool

Package general
Module images

Plot profiles with axis label conversions
include "imageprofilesupport.g"

Constructors

imageprofilesupport	Construct an imageprofilesupport tool

Functions

clearplotter	Clear the plotter
disablespectralrefmenu	Disables spectral reference change menu
done	Destroy this imageprofilesupport tool
getabcissa	Recover the profile abcissa(s)
getabcissaunit	Recover the profile abcissa unit
getabcissaunits	Recovers all the possible abcissa units
getcurrentabcissa	Recover the plotted profile abcissa(s)
getdoppler	Recover the profile doppler type
getisabs	Is the current abcissa unit absolute ?
getmask	Recover the profile mask
getoffsetdoppler	Recovers the offset doppler
getoffsetvalue	Recovers the offset value
getordinate	Recover the profile ordinate
getordinateunit	Recover the profile ordinate unit
getrefframe	Get current reference frame
getx	Get abcissa coordinate of cursor
getxy	Get abcissa and ordinate coordinate of cursor
hasplotter	Has the plotter been made yet ?
hasprofile	Has a profile been set yet ?
insertoffset	Set a new offset
makeabcissa	Make the abcissa
makemenus	Make the abcissa menus
makeordinate	Make an ordinate
makeplotter	Make the plotter
npoints	Return the number of points in the current profile
nprofiles	Return the number of profiles
plot	Plot the profile
plotfile	Saves the plot as a Table
plotter	Recover the plotter
point	Mark the specified coordinate with a symbol
postscript	Generate a postscript file of the plot
setabcissaunit	Set abcissa unit
setcoordinatesystem	Set a new coordinate system
setnoprofile	Reset number of profiles to zero
setordinate	Set an ordinate
setordinateunit	Set ordinate unit
setplotter	Set a new plotter
setprofile	Set a profile
setprofileaxis	Set the profile axis
setspectralref	Set new Spectral reference

settitle	Set plot title with fancy colors
setyrangecallback	Set ordinate data range via a callback function
which	Return indices of current plotted profiles

The imageprofilesupport `tool` is not meant as a general purpose high-level user `tool`. It is meant for use within high-level applications.
This `tool` allows you to plot a 1-D profile with built-in abcissa label coordinate conversions. For example, you can use it to plot a velocity profile and then select the abcissa units to be say km/s or GHz. You can also convert betwreen reference frames (for Spectal coordinates).
You can accumulate many profiles and display them together. You may plot all or selected profiles.
It has two construction modes. By default, `multiabcissa=F` which means there is one abcissa for one or more ordinates. In this mode, you can only use functions `makeabcissa`, `makeordinate` and **setordinate** to generate the profiles.
When `multiabcissa=T`, it is possible to have profiles which have distinct abcissas (including possibly different numbers of points). In this mode, you can only use function **setprofile** to generate the profiles.
You can optionally choose to have an abcissa offset entry box presented. If you do, it enables you to offset the spectra by some value. In particular, for Spectral axes, you can apply a velocity offset so that the spectrum is presented in the rest frame of the line.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
-
- p := imageprofilesupport(csys=csys,      # imageprofilesupport tool
                           shp=shp,
                           widgetset=dws)
-
- axis := 3                             # Profile axis
- p.setprofileaxis(axis)
- p.makemenus()                            # Unit menus go into pgplotter
-
- pos := csys.referencepixel()             # Position in image
- p.makeabcissa(pos)                       # Generate abcissa along profile axis
-
- blc := pos; blc[3] := 1                  # Region to extract profile
- trc := pos; trc[3] := shp[3]
- r := drm.box(blc, trc)
```

```
- id1 := p.makeordinate (im, r)           # Generate profile ordinate
-
- p.plot()                                # Plot all profiles
-
- r := drm.box()                          # New region to extract profile from
- id2 := p.makeordinate (im, r)           # Generate second profile ordinate.
  # Non-profile axes are averaged
- p.plot()                                # Plot all profiles
- p.plot(id2)                             # Plot second profile
```

In this example, we plot a profile from an image along the third axis. The axis label menus embedded in the specified frame can be used to redraw the plot with different axis labels.

Events
This `tool` emits six events.

- unitchange: the user has selected a new abcissa unit. The value of the event is the new unit (a string).
- absrelchange: the user has selected a new abs/rel from the abcissa menu. The value of the event is T (for absolute) or F (for relative).
- dopplerchange: the user has selected a new doppler type from the abcissa menu. The value of the event is the new doppler type (a string).
- spectralrefchange: the user has selected a new spectral reference from the menu. The value of the event is the new reference (a string).
- offsetvaluechange: the user has selected a new offset value. The value of the event is the new offset (a quantum).
- offsetdopplerchange: the user has selected a new doppler offset. The value of the event is the new offset doppler (a string).

imageprofilesupport - Constructor

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Construct an imageprofilesupport tool

Synopsis
imageprofilesupport(csys, shp, widgetset, multiabcissa, offset)

Arguments

csys	in	Coordinate System Allowed:Coordsys tool
shp	in	Shape of associated image Allowed:Vector of integer
widgetset	in	The widgetset to use for the GUI Allowed:A widgetserver tool Default: dws
multiabcissa	in	Allow multiple abcissas Allowed:T or F Default: F
offset	in	Show abcissa offset entry ? Allowed:T or F Default: F

Returns
imageprofilesupport tool

Description
This constructs the **imageprofilesupport** `tool`.

Example

```
- im := image('ngc1313.cube')
- csys = im.coordsys()
- p := imageprofilesupport(csys=csys, shp=im.shape())
```

---

imageprofilesupport.clearplotter - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Clear the plotter

Synopsis
clearplotter()

Returns
T

Description

This function clears the plotter. It also resets the internal abcissa and ordinate ranges (to empty) which is a trigger to ensure the axes are redrawn.

Example

```
- im := image('ngc1313.cube')             # Open image
- csys := im.coordsys()                   # Coordinate system
- shp := im.shape()                       # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(3);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- p.plot()
- p.clear()
```

---

imageprofilesupport.disablespectralrefmenu - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Disables spectral reference change menu

Synopsis
disablespectralrefmenu()

Returns
T

Description

This function disables the spectral reference conversion menu so that the user can no longer make reference conversions.

---

imageprofilesupport.done - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Destroy this imageprofilesupport tool

Synopsis
done()

Returns
Bool or fail

Description
This function destroys the imageprofilesupport `tool`.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.done()
- p
F
```

---

imageprofilesupport.getabcissa - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recover the profile abcissa(s)

Synopsis
getabcissa(which)

Arguments

which	in	Abcissa index Allowed:Integer Default: 1

Returns
Record

Description
This function recovers a record holding the specified abcissa. There is only once abcissa unless you are in multi-abcissa mode (set at contruction time). There are a variety of subfields such as 'pixel', 'nativeworld', and 'currentworld' which hold the abcissa in a variety of formats.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
#
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()             # Position in image
- p.makeabcissa(pos)                       # Generate abcissa along profile axis
- abc := p.getabcissa()
- field_names(abc)
pixel nativeworld currentworld name nativeunit currentunit currentveltype currentabs
```

---

imageprofilesupport.getcurrentabcissa - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recover the plotted profile abcissa(s)

Synopsis
getcurrentabcissa(which)

Arguments

which	in	Abcissa index Allowed:Integer Default: 1

Returns
Record

Description
This function recovers the current plotted abcissa.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
#
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()             # Position in image
- p.makeabcissa(pos)                       # Generate abcissa along profile axis
- abc := p.getcurrentabcissa()
- print shape(abc)
[64]
```

---

imageprofilesupport.getabcissaunit - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recover the profile abcissa unit

Synopsis
getabcissaunit()

Returns
String

Description
This function recovers the current abcissa unit.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- p.getabcissaunit()
GHz
```

---

imageprofilesupport.getabcissaunits - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recovers all the possible abcissa units

Synopsis
getabcissaunits()

Returns
Vector of strings

Description
This function recovers all of the possible abcissa units that the user might select from.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- p.getabcissaunits()
GHz MHz kHz Hz km/s m/s pix
```

---

imageprofilesupport.getdoppler - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recover the profile doppler type

Synopsis
getdoppler()

Returns
String

Description
This function recovers the current doppler type. If the profile axis is not spectral, the velocity type menu will be disabled. Regardless, this function will still return the current menu choice.

Example

```
- im := image('ngc1313.cube')            # Open image
- csys := im.coordsys()                  # Coordinate system
- shp := im.shape()                      # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- p.getdoppler()
radio
```

---

imageprofilesupport.getoffsetvalue - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recovers the offset value

Synopsis
getoffsetvalue()

Returns
Quantum

Description
This function recovers the current abcissa offset.

---

imageprofilesupport.getoffsetdoppler - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recovers the offset doppler

Synopsis
getoffsetdoppler()

Returns
String

Description
This function recovers the current abcissa offset doppler.

---

imageprofilesupport.getisabs - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Is the current abcissa unit absolute ?

Synopsis
getisabs()

Returns
T or F

Description
This function returns T if the abcissa is currently showing an absolute coordinate. Else it returns F.

Example

```
- im := image('ngc1313.cube')             # Open image
- csys := im.coordsys()                   # Coordinate system
- shp := im.shape()                       # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- p.getisabs()
T
```

---

imageprofilesupport.getmask - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recover the profile mask

Synopsis
getmask(which)

Arguments

which	in	ordinate index Allowed:Integer Default: 1

Returns
Vector of Bool

Description

This function recovers the mask for the specified profile. The mask (T for good pixels, F for bad pixels) is created for you by function makeordinate and created by you explicitly with function setordinate.
Masked pixels are not plotted.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
-
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- idx := p.makeordinate (im, r)
```

```
- p.getmask(idx)
[T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T
T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T]
```

---

imageprofilesupport.getordinate - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recover the profile ordinate

Synopsis
getordinate(which)

Arguments

which	in	ordinate index Allowed:Integer Default: 1

Returns
Record

Description
This function recovers a record holding the ordinate for the specified profile. There are two subfields: 'unit' (string) and 'data' (vector float) in the record holding the ordinate.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
-
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
```

```
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- idx := p.makeordinate (im, r)
- o := p.getordinate(idx)
- print o.unit
Jy/beam
- print o.data
[0.0692182779 0.313889951 0.221464232 -0.339388996 0.196267739 0.00308039412 0.145770207
0.083701618 -0.0170051232 -0.0674641132 -0.147130117 -0.0748230442 -0.0087018162
0.042476207 0.0843885541 0.0214372594 0.0204261355 -0.0142198177 -0.150894299 -
0.292182982 -0.287655115 -0.220203012 -0.311543584 -0.696057141 -0.594372749
-0.591924787 -0.640231609 -0.751704574 -0.889986157 -0.973421454 -0.9192608
-0.853011549 -1.01908207 -1.15568936 -1.04332018 -0.943786085 -0.671383262
-0.398980319 -0.274277657 -0.602319181 -0.813411593 -0.665026307 -0.283517033
-0.198741049 -0.649372399 -0.961182058 -0.79399693 -0.628294706 -0.550167799 -0.513776541
-0.453341246 -0.401189208 -0.362751096 -0.153813794 -0.0688151047 -0.11182557 -0.106346995
-0.0373723656 -0.0238029882 0.014433424 -0.106619254 -0.173805982 -0.253587902 -0.561424077]
```

---

imageprofilesupport.getordinateunit - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recover the profile ordinate unit

Synopsis
getordinateunit()

Returns
String

Description
This function recovers the ordinate unit.

Example

```
- im := image('ngc1313.cube')
```

```
- csys := im.coordsys()
- shp := im.shape()
-
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- p.makeordinate (im, r)
- p.getordinateunit()
Jy/beam
```

---

imageprofilesupport.getrefframe - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Get current reference frame

Synopsis
getrefframe()

Returns
String

Description

Get the current reference frame of the profile. If you are yet to set the profile axis, this returns an empty string. Otherwise it is the selected refererence frame (for Spectral coordinates), or the native reference frame (for Direction coordinates), or blank (other coordinates).

---

imageprofilesupport.getx - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Get abcissa coordinate of cursor

Synopsis
getx(color)

Arguments

color	in	Color of location marker Allowed:PGPlot color (expressed as integer index) Default: 7

Returns
Float

Description
Get the abcissa coordinate of the cursor and mark the x-location with a vertical line.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- p.makeordinate (im, r)
- x := p.getx()
```

---

imageprofilesupport.getxy - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Get abcissa and ordinate coordinate of cursor

Synopsis
getxy(symbol, color)

Arguments

symbol	in	PGPlot symbol to mark location Allowed:Integer Default: 2 (cross)
color	in	PGPlot color of symbol Allowed:Integer Default: 1 (foreground)

Returns
Record

Description
Get the abcissa and ordinate coordinate of the cursor and mark the location. Returns a record with fields 'x' and 'y' (each holding a float).

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- p.makeordinate (im, r)
- xy := p.getxy()
```

---

imageprofilesupport.hasplotter - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Has the plotter been made yet ?

Synopsis
hasplotter()

Returns
T or F

Description
This function returns T if plotter has been created and F if it is yet to be made.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.hasplotter()
F
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- p.plot()
- p.hasplotter()
T
```

---

imageprofilesupport.hasprofile - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Has a profile been set yet ?

Synopsis
hasprofile()

Returns
T or F

Description
This function returns T if a profile has been created and F otherwise. The profile is considered to have been created after makeabcissa has been called. The profile is considered to have been destroyed when function setnoprofile has been called.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.hasprofile()
F
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- p.hasprofile()
T
- p.setnoprofile()
- p.hasprofile()
F
```

---

imageprofilesupport.insertoffset - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set a new offset

Synopsis
insertoffset(offset, doppler)

Arguments

offset	in	Offset Allowed:Quantum or quantum string Default: No change
doppler	in	Doppler Allowed:String Default: No change

Returns
T, F or fail

Description
This function sets a new offset. The plot is redrawn after.

---

imageprofilesupport.makeplotter - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Make the plotter

Synopsis
makeplotter(parent, size)

Arguments

parent	in	Parent frame for plotter Allowed:Tk agent Default: Standalone
size	in	Size of plotter Allowed:Vector of int Default: [295,215]

Returns
T or fail

Description
This function creates the plotter. If you supply a parent frame, a pgplotter

`tool` will be embedded in it. If you don't supply a parent frame, a standalone pgplotter `tool` will be created.
Calling this function is optional. If you don't call this function, then a call to plot will create a stand alone plotter (a pgplotter `tool`.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.makeplotter()
```

---

imageprofilesupport.makemenus - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Make the abcissa menus

Synopsis
makemenus(parent)

Arguments

parent	in	Parent frame for plotter Allowed:Tk agent Default: Use PGPlotter

Returns
T or fail

Description

This function creates three menus from which the user controls what form the abcissa labelling will take. These menus are to control whether the abcissa coordinate is absolute or relative, what the units are, and, if applicable (for

spectral profile axes) what the doppler type is. If you don't give the parent frame, the menus are embedded in the Pgplotter `tool` that is generated for plotting.
If you haven't yet specified the profile axis via function setprofileaxis. (maybe you don't know it when you have to create these menus), empty menus will be made.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(3);
- p.makemenus()
```

---

imageprofilesupport.makeabcissa - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Make the abcissa

Synopsis
makeabcissa(pixel)

Arguments

pixel	in	Pixel coordinate for abcissa Allowed:Vector of integer

Returns
T or fail

Description
This function creates the abcissa. You specify the absolute pixel coordinate for which the abcissa should be generated. It will take the full range of the shape (specified at construction) for the profile axis.

You must call function setprofileaxis before calling this function so that it knows which axis is the profile axis.

Example

```
- im := image('ngc1313.cube')              # Open image
- csys := im.coordsys()                    # Coordinate system
- shp := im.shape()                        # Shape of image
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
```

---

imageprofilesupport.makeordinate - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Make an ordinate

Synopsis
makeordinate(im, region, ci, ls, which)

Arguments

im	in	The image Allowed:Image tool
region	in	The region of interest Allowed:Region tool or naked record
ci	in	PGPlot color index of this ordinate when plotted Allowed:Integer Default: 1
ls	in	PGPlot line style of this ordinate when plotted Allowed:Integer Default: 1
which	in	Which ordinate index to set Allowed:Integer or unset Default: Next slot

Returns
Integer or fail

Description
This function creates an ordinate. You specify an Image `tool` and a Region `tool`. The image data in the region are averaged over all axes except the profile axis. The region may also be specified as a naked record with fields 'blc' and 'trc' holding vectors of absolute pixel coordinates.
This function returns an index identifying the ordinate number that you are displaying. You can display many profiles simultaneously this way. To reset the number of profiles to zero use function setnoprofile.
You must call function setprofileaxis before calling this function so that it knows which axis is the profile axis.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
-
- pos := csys.referencepixel()
- p.makeabcissa(pos)
-
- blc := pos; blc[3] := 1
- trc := pos; trc[3] := shp[3]
- r := drm.box(blc, trc)
- idx := p.makeordinate (im, r)
-
- p.plot(which=idx)
```

---

imageprofilesupport.npoints - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Return the number of points in the current profile

Synopsis
npoints()

Returns
Integer

Description
This function returns the number of points in the current profile. Will be zero if none yet made. It is function makeabcissa that really determines the number of points (the ordinate must conform).

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
-
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- p.npoints()
64
```

---

imageprofilesupport.nprofiles - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Return the number of profiles

Synopsis
nprofiles()

Returns
Integer

Description
This function returns the number of profiles stored.

---

imageprofilesupport.plot - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Plot the profile

Synopsis
plot(xautoscale, yautoscale, which)

Arguments

xautoscale	in	Autoscale abcissa ? Allowed:T or F Default: T
yautoscale	in	Autoscale ordinate ? Allowed:T or F Default: T
which	in	Ordinate index Allowed:Integer or unset Default: All

Returns
T or fail

Description
This function plots all or the specified ordinate profile.
Every time you call functions makeordinate or setordinate a new ordinate profile is added to the display. Each of these functions returns an index which you can use here to specify via argument `which` which profile you want to see. By default, all profiles available are displayed.
By default the abcissa and ordinates are autoscaled - the ranges are worked out from all of the current profiles available.
If you turn off x-autoscaling, then the new plot is drawn with the current abcissa range (e.g. from a zoom).
If you turn off y-autoscaling, then the new plot is drawn with the y-range provided via the call back function setyrangecallback is used. If there is no callback, then the current range will be used.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
-
- p := imageprofilesupport(csys=csys, shp=shp,
                           widgetset=dws)
-
- f1 := dws.frame()
- axis := 3
- p.setprofileaxis(axis);
- p.makemenus()
-
- pos := csys.referencepixel()
- p.makeabcissa(pos)
-
- blc := pos; blc[3] := 1
- trc := pos; trc[3] := shp[3]
- r := drm.box(blc, trc)
- id1 := p.makeordinate (im, r)                    # Profile 1
-
- r := drm.box()
- id2 := p.makeordinate (im, r)                    # Profile 2
-
- p.plot(which=id1)                                # Plot profile 1
- p.plot(which=id2)                                # Plot profile 2
- p.plot()                                         # Plot all profiles
```

---

imageprofilesupport.plotter - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Recover the plotter

Synopsis
plotter()

Returns
pgplotter or pgplotwidget tool or fail

Description
This function recovers the underlying plotter `tool`. This will be either a PGPlotter (stand alone) or a PGPlotWidget (embedded) `tool`.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
-
- p := imageprofilesupport(csys=csys, shp=shp,
                           widgetset=dws)
-
- f1 := dws.frame()
- axis := 3
- p.setprofileaxis(axis);
- p.makemenus()
-
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- p.makeordinate (im, r)
- p.plot()
- pl := p.plotter()
```

---

imageprofilesupport.point - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Mark the specified coordinate with a symbol

Synopsis
point(x, y, x, errx, erry, symbol, color)

Arguments

x	in	Abcissa coordinate Allowed:Float
y	in	Ordinate coordinate Allowed:Float
x	in	Abcissa coordinate Allowed:Float
errx	in	Half width of x error bar Allowed:Float Default: None
erry	in	Half width of y error bar Allowed:Float Default: None
symbol	in	PGPlot marker symbol Allowed:Integer Default: 2 (cross)
color	in	PGPlot color index of symbol Allowed:Integer Default: 1 (foreground)

Returns
T

Description
This function marks the specified location with a symbol and optional error bars. The specified point is not retained in any draw lists so this is a one off draw.

---

imageprofilesupport.postscript - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Generate a postscript file of the plot

Synopsis
postscript(file, color, landscape)

Arguments

file	in	postscript file name Allowed:String
color	in	Use color ? Allowed:T or F Default: T
landscape	in	Use landscape or portrait Allowed:T or F Default: T

Returns
T or fail

Description
This function saves the current plot in a postscript file.

---

imageprofilesupport.plotfile - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Saves the plot as a Table

Synopsis
plotfile(file)

Arguments

file	in	file name Allowed:String

Returns
T or fail

Description
This function saves the current plot in an AIPS++ Table. The plot can then be reopened and new commands added or old commands changed.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys,
                           shp=shp,
                           widgetset=dws)
- f1 := dws.frame()
- axis := 3
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- p.makeordinate (im, r)
- p.plot()
- p.plotfile('save.plot')         # save
- p2 := pgplotter('save.plot')    # reopen
```

---

imageprofilesupport.setabcissaunit - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set abcissa unit

Synopsis
setabcissaunit(unit, doppler)

Arguments

unit	in	Abcissa unit Allowed:String
doppler	in	Abcissa Doppler type Allowed:String Default: unset

Returns
T

Description

This function sets the plot abcissa unit. You can only call this function after a call to makemenus. The specified unit must exist in the menu list. Any extant plot will be redrawn after this call. In general spectra will be plotted in the default RADIO doppler type when the abcissa is a velocity. This behavior can be changed by the doppler argument.

---

imageprofilesupport.setcoordinatesystem - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set a new coordinate system

Synopsis
setcoordinatesystem(csys, shp, resetref)

Arguments

csys	in	New coordinate system Allowed:Coordsys tool
shp	in	Shape of associated image Allowed:Vector of integer
resetref	in	Reset Reference Frame to Native of new Coordinate System Allowed:T or F Default: F

Returns
Integer or fail

Description

This function sets a new coordinate system and associated image shape. This function will update the x-axis menus appropriately.
After calling this function, you must remake the abcissa and ordinate profiles (functions makeabcissa and makeordinate or setordinate).
The argument `resetref` is used to reset any reference frame selection (in the GUI) back to the native one contained in the new coordinate system (or any

extra frame conversion currently set in the supplied coordinate system). Otherwise, the GUI selection is left as it is. For example, the user might have asked to convert a Spectral coordinate to TOPO. If `resetref=F` TOPO is left selected and conversions are made to it. If `resetref=T` then, if the new coordinate system has LSRK say as the native frame, then LSRK is now selected in the GUI and no extra conversions are made.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- axis := 3;
- p := imageprofilesupport(csys=csys, shp=shp)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- p.makeordinate(im, r)
- p.plot()
-
- im2 := image('ngc1399.cube')
- csys2 := im2.coordsys();
- shp2 :=  im2.shape();
- axis2 := 1;
- p.setprofileaxis(axis2)
- p.setcoordinatesystem(csys2, shp2)
- pos := csys.referencepixel()
- p.makeabcissa(pos);
- p.makeordinate(im2, r)
- p.plot()
```

---

imageprofilesupport.setnoprofile - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Reset number of profiles to zero

Synopsis
setnoprofile()

Returns
T

Description
This function resets the number of accumulated profiles to zero.

---

imageprofilesupport.setordinate - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set an ordinate

Synopsis
setordinate(data, mask, err, ci, ls, which)

Arguments

data	in	The ordinate data Allowed:Vector of float
mask	in	The ordinate mask Allowed:Vector of Bool Default: All good
err	in	The ordinate error Allowed:Vector of float Default: None
ci	in	PGPlot color index of this ordinate when plotted Allowed:Integer Default: 1
ls	in	PGPlot line style (pos) or graph marker (neg) of this ordinate when plotted Allowed:Integer Default: 1
which	in	Which ordinate index to set Allowed:Integer Default: Next slot

Returns
Integer or fail

Description

This function sets an ordinate directly, whereas function makeordinate generates it for you from an image.
If you don't set the mask it is assumed to be all T (good).
This function returns an index identifying the ordinate index that you are setting. You can display many profiles simultaneously this way. To reset the number of profiles to zero use function setnoprofile. You can ovewrite an ordinate by specifying `which`.
If argument `ls` is positive, then the points are drawn as a line with PGPLOT linestyle `ls`. If it is negative, then the points are drawn as graph markers with PGPLOT marker value `abs(ls)`.
You must call function setprofileaxis before calling this function so that it knows which axis is the profile axis.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
-
- pos := csys.referencepixel()
- p.makeabcissa(pos)
-
- blc := pos; blc[3] := 1
- trc := pos; trc[3] := shp[3]
- r := drm.box(blc, trc)
- local data, mask
- im.getregion(data, mask, region=r, dropdeg=T)
- idx := p.setordinate (data, mask, 1)
- p.plot()
```

---

imageprofilesupport.setprofile - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set a profile

Synopsis
setprofile(abcissa, ordinate, orderr, mask, unit, doppler, ci, ls, which)

Arguments

abcissa	in	The abcissa Allowed:Vector of float
ordinate	in	The ordinate Allowed:Vector of float
orderr	in	The error in the ordinate Allowed:Vector of float Default: None
mask	in	The mask Allowed:Vector of Bool Default: All good
unit	in	The abcissa unit Allowed:'pix' or valid unit Default: 'pix'
doppler	in	The doppler type Allowed:'radio', 'optical', 'true' Default: 'radio'
ci	in	PGPlot color index of this ordinate when plotted Allowed:Integer Default: 1
ls	in	PGPlot line style of this ordinate when plotted Allowed:Integer Default: 1
which	in	Which profile index to set Allowed:Integer Default: Next slot

Returns
Integer or fail

Description

This function sets a profile, both abcissa and ordinate directly. It allows spectra with distinct abcissas to be plotted together. Such simultaneous profiles can have different numbers of points as well.

This function is only available if you construct with the argument `multiabcissa=T`. Otherwise, you are restricted to one abcissa with one or more ordinates.
If you don't set the mask it is assumed to be all T (good).
This function returns an index identifying the profile index that you are setting. You can display many profiles simultaneously this way. To reset the number of profiles to zero use function setnoprofile. You can ovewrite a profile by specifying `which`.
You must specify the unit of the abcissa vector which must always be in absolute coordinates. You may use absolute pixels (i.e. those of the construction coordinate system) by setting `unit='pix'`. Otherwise `unit` can be any valid world unit consistent with the coordinate type of that axis. If the profile axis pertains to a spectral coordinate, you may use a velocity (e.g. 'm/s') or frequency (e.g. 'MHz') unit. Also for spectrla abcissas, if you specify a velocity unit, you shouyld specify the doppler type.
You can mix abcissa specification units with subsequent calls.
You must call function setprofileaxis before calling this function so that it knows which axis is the profile axis.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, multiabcissa=T)
- p.setprofileaxis(axis);
- p.makemenus()
-
- id := [];
- x := 1:100
- y := (1:100)/100.0;
- id[1] := p.setprofile (abcissa=x, ordinate=y, unit='pix');
--
- x := 120:200
- n := length(x);
- y := (1:n) / n;
- id[2] := p.setprofile (abcissa=x, ordinate=y, unit='pix');
- p.plot()
```

---

imageprofilesupport.setplotter - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set a new plotter

Synopsis
setplotter(plotter)

Arguments

plotter	in	New plotter Allowed:Pgplotter tool

Returns
T or fail

Description

This function allows you to set a plotter with one of your own creation. It must be a pgplotter `tool`.
After calling this function, you must remake the abcissa and ordinate profiles (functions makeabcissa and makeordinate or setordinate).

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- axis := 3;
- p := imageprofilesupport(csys=csys, shp=shp)
- p.setprofileaxis(axis);
- p.makemenus()
- pos := csys.referencepixel()
- p.makeabcissa(pos)
- r := drm.box()
- p.makeordinate(im, r)
- p.plot()
-
- pl := pgplotter()
- p.setplotter(pl)
- p.makemenus()
- p.makeabcissa([1,1])
- p.setordinate(data2)
- p.plot()
```

imageprofilesupport.setprofileaxis - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set the profile axis

Synopsis
setprofileaxis(axis)

Arguments

axis	in	The profile axis Allowed:Integer Default: -1

Returns
T

Description

This function sets the profile axis. The default, -1, means that no axis is effectively set. The x-unit menus will be updated appropriately, if they have already been made.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
```

---

imageprofilesupport.setordinateunit - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set ordinate unit

Synopsis
setordinateunit(unit)

Arguments

unit	in	Ordinate unit Allowed:String

Returns
T

Description

This function sets the ordinate unit. You need to do this if you are using setordinate rather than makeordinate to set the ordinate.

---

imageprofilesupport.setspectralref - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set new Spectral reference

Synopsis
setspectralref(value)

Arguments

value	in	The new spectral reference label Allowed:String

Returns
T, F or fail

Description

This function lets you set the spectral reference directly (rather than selecting the menu item interactively). When you do so, a **spectralrefchange** event is emitted. Returns F if string is invalid.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- p.setprofileaxis(axis);
- p.makemenus()
- p.setspectralref ('TOPO')
```

---

imageprofilesupport.setyrangecallback - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set ordinate data range via a callback function

Synopsis
setyrangecallback(callback)

Arguments

callback	in	Callback function Allowed:Glish function

Returns
T

Description

In function plot you may turn off profile ordinate autoscaling. If you do so, then the new display is drawn with the current ordinate range if there is one.

If there is no current ordinate range available, the ordinate range is provided via this call back function. The callback function must return a vector of two floats, the ordinate minimum and maximum.

---

imageprofilesupport.settitle - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Set plot title with fancy colors

Synopsis
settitle(text, ci)

Arguments

text	in	Title text Allowed:Vector of strings
ci	in	PGPLot colour index for each string Allowed:Vector of integers

Returns
T

Description

This function allows you to write a title with different words in different colors.

Example

```
- im := image('ngc1313.cube')
- csys := im.coordsys()
- shp := im.shape()
- p := imageprofilesupport(csys=csys, shp=shp, widgetset=dws)
- t := "A silly title";
- ci := [1,2,7];       # Reg Green Yellow
- p.settitle(t, ci)
```

---

imageprofilesupport.which - Function

Package	**general**
Module	**images**
Tool	**imageprofilesupport**

Return indices of current plotted profiles

Synopsis
which()

Returns
unset or vector of int

Description

This function allows returns the profile indices for the profiles currently plotted.

1.2.21 imageprofilesupporttest - Function

Package	**general**
Module	**images**

Run test suite

Synopsis
imageprofilesupporttest(which, destroy)

Arguments

which	in	Which test to run Allowed:Integer or vector of integer Default: All tests
destroy	in	Destroy plotters Allowed:T or F Default: T

Returns
T or fail

Description
Run the test suite.
