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