argument is a componentlist tool. If so it returns True, otherwise it returns False.

Arguments

tool	The variable that you wish to test	
	Allowed	Any glish variable
	Default	no default

Returns
T or F, indicating if the argument is a componentlist tool or not.

Example

```
include 'componentlist.g'
if (is_componentlist(cl)) {
  cl.simulate(2);
} else {
  fail 'Not a componentlist';
}
```

This example shows how you can use this function to determine if the supplied variable is a componentlist.

1.3.3 componenteditor - Tool

Package	synthesis
Module	componentmodels

A graphical user interface for displaying & editing a component.
include "componenteditor.g"

Constructors

componenteditor	Create an componenteditor tool

Functions

dismiss	Shut down the GUI

done	Shut down the componenteditor tool
get	get the component currently being edited
gui	Start up the GUI
use	Supply a component to be edited
which	return the index of this component

Description

The componenteditor is a graphical user interface that allows the user to view & modify components. To use it you need to be in a windows environment ie., the display environment must be set prior to starting up AIPS++.
The componenteditor can be in one of two states, view mode or edit mode.

View Mode The component editor always starts up in view mode. In this mode you cannot change any of the fields in the component. View mode is useful for browsing a component. To switch to Edit mode you press the Edit button just below the menu bar.

Edit Mode In edit mode you can modify the component. Be aware that the editor reads all input from the screen in edit mode so it is important to display all numbers with sufficient precision. To switch to back to view mode press the View button just below the menu bar.

You can always tell which mode you are in because the entry field backgrounds are grey (and input is disabled) while in view mode, but white in edit mode.
There are a variety of buttons and pull-down menus for specifying the units and reference frames for various aspects of the component. In edit mode these specify how the numerical value is to be interpreted. In view mode they control the units, reference frame and precision of the displayed numerical values. Buttons that do not make sense in view mode, eg. those to change a components shape, are disabled in view mode.
When editing fields in this GUI it is suggested you press return at the end of each field. This will force the GUI to parse the current line and redisplay what it has read. However some effort is made to ensure that displayed fields are synchronised with the internal data even if you do not press return. This is down by redisplaying the data whenever a button is pressed. However the data is *not* redisplayed on every keystroke.
The componenteditor window is split up into seven fundamental frames. I'll describe these in the following sections.

Menu bar

The menu bar is the topmost part of the componenteditor GUI. It currently contains two pull down menus, ‘‘Show’’ and ‘‘Help’’.

Show This pull-down menu allows you to control which aspects of a component are displayed by the editor. It allows you to display or hide the label, flux, shape and spectrum frames.

This menu indicates which frames are being displayed with a red check box. Because of a bug in tcl/tk this indication can get out of synchronisation with what is being displayed. This occurs when two or more editors are being used simultaneously.

Help This pull-own menu contains functions which will drive you www browser to relevant sections of the AIPS++ reference manual. These sections are:

- The componenteditor documentation (thats what you are reading now)!
- The componentlist documentation.
- The module documentation for components.
- The contents page of the AIPS++ reference manual.

The about function will pop up a window displaying some information about the version of AIPS++ you are currently using.

Mode

The mode line is just below the menu bar. It contains the View and Edit buttons which switch the editor between these two modes. The current mode is indicated by a red check mark next to one of these buttons.

Label

The label frame is just below the mode line and displays the label string of the component. While only 24 characters are displayed there is no limit, aside from computer memory, on how long the label string can be. To see the end of a long label string you could make the componenteditor window wider using your window manager. Alternatively you could scroll through the string using the arrow keys. The text entry widget used to display the label also supports some emacs like control sequences for cursor movement eg., control-a for beginning of line, and control-e for end of line.

Flux

The flux frame displays the flux of the component. By default all polarisations of the component are displayed using its units and polarisation representation.
To display or set the flux in a different polarisation representation use the pull-down menu to the left of this frame. If the linear (XY) or circular (RL) representations are chosen then the displayed values are complex, otherwise the displayed values are real. You can also choose to hide all the polarisation components except the total intensity.
To display or set the flux units the buttons on the right of this frame are used. Allowable units are Jansky (Jy) and Westerbork Units (WU). You can also display the polarised flux as a percentage of the total intensity, when the polarisation representation is Stokes.
If the polarisation representation is ‘‘Stokes’’ the entered flux values are read as floating point numbers. But if the polarisation representation is linear or circular the entered flux values are read as complex numbers. They are displayed similarly.

Shape

The shape frame displays information about the spatial structure of the component. It always shows the reference direction of the component, the reference frame and the shape type. If the shape type in not a point there will be additional fields containing the other parameters relevant to specific shape.
The reference direction is the for all currently available component shapes the direction of the centre of the component. It can be displayed in a variety of units, formats and precisions using the pull-down menu to the right of the displayed values. Because the direction values can be entered in a variety of different formats entering these strings is more prone to error. Hence I recommend you always press return after entering a value to see how the editor has interpreted your string. The previously mentioned note about loss of precision when in edit mode is particularly relevant in the direction fields. If sub-arcsecond precision is important to you set the RA units to 'H:M:S.s' and the declination units to 'D.M.S.s' prior to entering edit mode.
The reference frames allowed for the direction are currently limited to those that do not require additional information about when are where the observation took place. Currently this is 'B1950', 'J2000' & 'Galactic' frames.

Three component shapes are currently supported are 'point', 'Gaussian' & 'disk'. For the Gaussian and disk shapes there are three additional parameters, namely the major axis, minor axis and position angle required. Fields displaying these parameters will be displayed if the shape is not a point shape.
The major axis is always larger than (or equal to) the minor axis and the position angle is the angle between the major axis and the North-South axis. The north-south axis is defined using the current reference frame and hence a component with a position angle of zero degrees in the galactic reference frame will have a very different position angle if the component is converted to the J2000 frame. As with other fields in the componenteditor the units are specified using pull-down menus to the right of the displayed values.

Spectrum

The spectrum frame displays information about the spectral variation of the component. Currently there are two models used to describe the spectral variation, namely 'Constant', and 'Spectral Index'.
If the spectrum is assumed to be constant then the concept of a reference frequency is irrelevant. Hence it is not displayed.
If the spectral model used is a spectral index then a reference frequency, frequency reference frame and spectral index need to be specified. The units for the reference frequency are specified by a pull-down menu to the right of the value.
The frequency frame is specified using a pull-down menu just below the reference frequency. Currently no attempt is made to convert the reference frequency between different frames when in view mode. Hence you cannot change the frame in view mode.
To fully specify the spectral index of a polarised component four indices are used. These indices are applied to the I,Q,U&V components of the flux separately. If you are not interested in the spectral index of the polarised flux you can choose to display the spectral index of just the I component using a pull-down menu to the right of the I index value.

Actions

There are two buttons at the bottom of the componenteditor which provide quick access to common operations. To the left is the ''Undo'' button and to the right is the ''Dismiss'' button.
The ''Dismiss'' button makes the component editor window go away. Use this button when you have finished making your changes. Note that the componenteditor tool has not gone away, just its GUI

elements. In particular you can still access the (possibly edited) component it contains using the tool functions described below.
The ''Undo'' button will replace the currently displayed component with the one the component editor was started with[4]. The undo button is disabled if the editor knows that the currently displayed component is identical with its initialised one. If the undo button is enabled then the editor believes that the component has been modified. This may not always be correct as the editor does not compare numbers to determine if a component has been modified, it only determines if you had the potential to modify the component.

Example

```
include 'componenteditor.g'
ce := componenteditor();
ce.gui();
# After some pointing and clicking in the GUI
mycomponent := ce.get();
ce.done();
```

In this example I show how to use the componenteditor GUI as a separate tool that is not invoked by the componentlist tool or the componentlist GUI. The first line is the obligatory include line. The second constructs the tool. It initialises itself with a default component but does not start up the GUI until the the third line. I assume the user clicked on the dismiss button to get rid of the GUI window. The edited component is extracted from the componenteditor tool using the get tool function. All the memory used by the componenteditor is released in the last step. At this point the tool no longer exists.

componenteditor - Constructor

Package	synthesis
Module	componentmodels
Tool	componenteditor

Create an componenteditor tool

[4]To be more specific it replaces the component with the one that was last supplied to it with the use tool function.

Synopsis
componenteditor()

Description
This function creates a componenteditor tool. The componenteditor initialises itself with a default component but does not start up the GUI. Use the GUI tool function described below to start up the GUI. The componenteditor always starts up in view mode.

Returns
A componenteditor tool

---

componenteditor.use - Function

Package	synthesis
Module	componentmodels
Tool	componenteditor

Supply a component to be edited

Synopsis
use(component, which)

Description
This tool function supplies a component for the GUI to edit. A component is a Glish record with the fields and structure described in the componentlist tool.
The optional which argument is to allow multiple componenteditors to simultaneously edit different components in a componentlist. It contains the index of the component in the main list. This index can be obtained at a later time using the which tool function described below.
This scheme will break down if the main list has been sorted or components have been added or deleted while an editor is editing a copy of a component.
The which argument can be totally ignored if the componenteditor is not being used in conjunction with a parent list, or if you are separately keeping track of which component goes where. Otherwise it should be an integer between 1 and the parent list length

Arguments

component	The component to be viewed and/or edited	
	Allowed	Any Glish record with the appropriate fields
	Default	no default
which	The index of the component in a parent table	
	Allowed	Any integer (see description)
	Default	0

Returns
T

---

componenteditor.gui - Function

Package	synthesis
Module	componentmodels
Tool	componenteditor

Start up the GUI

Synopsis
gui()

Description
Start up the graphical user interface for this tool. The graphical user interface is not started generated when the tool is created. This separates the creation and destruction of the tool (using the constructor and the done tool function), with the creation and destruction of the GUI (using the GUI and dismiss tool functions). This separation allows the tool to be initialised, using the use tool function, prior to starting up the GUI.

Returns
T or fail if the window could not be created

---

componenteditor.dismiss - Function

Package	synthesis
Module	componentmodels
Tool	componenteditor

Shut down the GUI

Synopsis
dismiss()

Description
This tool function will make the GUI disappear from the screen. However the tool has not been deleted and hence the component that was being edited can still be obtained using the get tool function.

Returns
T

---

componenteditor.get - Function

Package	synthesis
Module	componentmodels
Tool	componenteditor

get the component currently being edited

Synopsis
get()

Description
This tool function returns the component currently being edited by this tool. The component is constantly kept up to date with the graphical representation seen by the user. Hence it can quite safely be used regardless of the interim state of the GUI.

Returns
A Glish record representing the edited component

---

componenteditor.which - Function

Package	synthesis
Module	componentmodels
Tool	componenteditor

return the index of this component

Synopsis
which()

Description
This tool function returns index of the (possibly edited) component within the a parent componentlist. This index is used to tag which component is being edited so that it can be replaced once the user has finished editing it.

Returns
The integer supplied to it with the use argument

---

componenteditor.done - Function

Package	synthesis
Module	componentmodels
Tool	componenteditor

Shut down the componenteditor tool

Synopsis
done()

Description
This tool function removes all traces of the componenteditor tool. It removes the GUI from the screen, deletes the component inside the editor and removes all the tool functions. It frees up as much memory as possible.

Returns
T

1.4 imager - Module

Module for synthesis and single dish imaging
include "imager.g"

Description

imager provides a unified interface for synthesis and single dish imaging including deconvolution starting from a MeasurementSet.

What imager **does:**

Standard synthesis and single dish imaging imager does nearly all types of synthesis and single dish imaging, including dirty images, point spread functions, deconvolution, combination of single dish and synthesis, spectral imaging, polarimetry, wide-field imaging, mosaicing, holography, near-field imaging, tracking moving objects, *etc.*. As a result of this extensive range of capabilities, it can be complicated to use, especially for the more esoteric forms of imaging.

Fine scaled tools Rather than present one operation to process data from visibilities to a restored, deconvolved image, imager contains a number of distinct, separate tool functions that allow careful tuning of the processing. For example, the weights used in imaging (the IMAGING_WEIGHT column in the MeasurementSet), can be altered via a number of tool functions (weight, filter) and inspected via a plotting tool function plotweights. Similarly, the deconvolution and restoration steps are separate, allowing user control of each step. It is our intention that other imaging tools may be built on top of imager: see, for example, imagerwizard, which also has the side-benefit that it displays the imager (and other tools) commands as they are executed.

Spectral imaging imager can perform either spectral imaging or frequency synthesis (producing either an image with each channel imaged independently or with some or all channels summed together). Channels may be selected in a number of ways, either as channels or as velocities. Also a continuum model image may be subtracted prior to making a cube.

Many different deconvolution algorithms imager is rich in deconvolution algorithms, including a number of clean variants, maximum entropy, non-negative least squares, and the pixon algorithm.

Mixing of deconvolution functions Since the deconvolved images are calculated and kept purely as images (rather than lists of clean components), deconvolution functions may be mixed as desired. Thus, one may use NNLS to deconvolve part of the Stokes I of an image, and then use CLEAN to deconvolve another part of all polarizations in the image. Note that a list of clean components is not available.

Ability to fix model images In a multifield deconvolution, it is possible to specify that some fields are not to be deconvolved, using the fixed argument of clean.

Single dish imaging imager can process single dish observations much as it does synthesis images. To make images with no deconvolution, use the makeimage function. This allows construction of traditional single dish images and holography images. To deconvolve images, just use the ''multifield'' deconvolution algorithms in clean and mem. You will want to set the gridmachine in setoptions to 'sd'.

Combination of single dish and synthesis data If the single dish and interferometer data are in the same MeasurementSet, then imager can perform a joint deconvolution using ''multifield'' deconvolution algorithms in clean and mem. You will want to set the gridmachine in setoptions to 'both'. You can change the relative weighting of synthesis and single dish data by using setsdoptions. If the single dish and synthesis data cannot be combined into one MeasurementSet then you can still use the feather function to combine already deconvolved images.

Multi-field processing imager can be run on any number of images, each of which can have any direction for the phase center. All coordinate transformations are done correctly. Using the measures system, these fields may be given moving positions (such as the Sun using dm.direction('sun') to specify the phase center) or positions in strange coordinates (such as Supergalactic using *e.g.* dm.direction('supergal', '0d', '0d') as well as the more conventional representations (*e.g.* dm.direction('b1950', '12h26m33.248000', '02d19m43.290000') specifies the coordinates of the core of 3C273). Note that for some coordinate systems a location must be supplied via the setoptions tool function. For example, one can put an image at a specific azimuth and elevation (*e.g.* dm.direction('azel', '67.4d', '5.23d')) at the VLA imgr.setoptions(location=dm.observatory('VLA')). Phase rotation will be automatically calculated to track in azimuth-elevation.

Wide-field imaging imager can perform wide-field imaging as needed to overcome the non-coplanar baselines effect for the VLA and other non-coplanar arrays.

Mosaicing imager can perform clean-based or mem-based mosaicing of many pointings into one image, using variants of the multi-field algorithms.

Processing of component lists Discrete components (not the same as clean components!) can be represented by componentmodels. A componentlist can hold any number of components. The components are subtracted from the visibility data before construction of an image. For high precision imaging, it is recommended that components be used for bright sources since the Fourier Transform of components avoids the limitations of the gridded transforms.

Joint deconvolution of Stokes IQUV imager can produce images of either I alone, or I, V or I, Q, U, V, deconvolving jointly as appropriate. The point spread function is constrained to be the same for all processed polarizations so asymmetric u,v coverage is not allowed.

Production of complex images imager can produce dirty or residual images or point spread functions in the original data representation (*e.g.* RR,RL,RL,LL or XX,XY,YX,YY.

Fine control and evaluation of visibility weighting Various tool functions for controlling the visibility weights are available (weight, filter) as well as tool functions for evaluating the effects of the weighting (plotweights, sensitivity, fitpsf). The Briggs algorithm for weighting of visibility data can be used (see weight and Dan Briggs' thesis).

Flexible windowing in the deconvolution Rather than use boxes to limit the region CLEANed, a mask image is used to constrain the region in which flux is allowed. There are various tool functions for making a mask image, including from regions and blc/trc specifications, and via thresholding the Stokes I image. In the Clark Clean, the mask is *soft*: it can vary between 0 and 1. Intermediate values of the mask bias against but do not rule out subtraction of clean components.

Non-Negative Least Squares Deconvolution This algorithm is very effective at producing high dynamic range images of moderately resolved sources (see Dan Briggs' thesis). It works on Stokes I alone so the recommended procedure is to CLEAN I, Q, U, V using clean and then use NNLS to refine the I part of the image using nnls.

Specification of arguments as measures A measure is a measured quantity with optional units, coordinates and reference frames. These are allowed in a number of circumstances. The advantage is that the user can specify arguments in very convenient form, and let the measures system do whatever conversion is required. For example:

Cell sizes These can be specified as a quantity (see the measures module).

```
imgr.setimage(cellx='7arcsec', celly='7arcsec')
```

Image center direction This must be specified as a direction (see the measures module).

```
imgr.setimage(phasecenter=dm.direction('j2000', '05h30m', '-30.2deg'))
imgr.setimage(phasecenter=dm.direction('gal', '0deg', '0deg'))
imgr.setimage(phasecenter=dm.direction('mars'))
imgr.setimage(phasecenter=image('myother.image').coordmeasures().direction);
```

Velocities These can be specified as radial velocities.

```
imgr.setimage(start=dm.radialvelocity('25km/s'),
              step=dm.radialvelocity('-500m/s'))
```

Position For construction of images in some coordinate frames (*e.g* azimuth-elevation) the position to be used in processing must be set:

```
imgr.setoptions(location=dm.observatory('ATCA'))
```

More choice in image size Any even image size will work, though to speed the FFT, it is advisable to use a highly composite number (one that has many factors). The advise function will calculate an acceptable number.

Integrated plotting Plots of visibility amplitude, weights (both point-by-point and gridded), uv coverage, and field and spectral window ids are available (plotvis, plotweights, plotuv, plotsummary).

Synchronous or Asynchronous processing Operations that take a substantial amount of time to run can be run in the background either by setting the global variable dowait:=F or by setting an argument *e.g.* imgr.clean(async=T). To retrieve a result, use the result tool function of defaultservers with the job number as the argument. For example:

```
- imgr:=imager('ss433.MS')
T
- imgr.setimage(cellx='0.05arcsec', celly='50marcsec', nx=256, ny=256,
  spwid=1:2, fieldid=1, stokes='IV')
T
- imgr.fitpsf()
1
# Wait for it to finish and then ask for the result:
- defaultservers.result(1)
[psf=, bpa=[value=42.7269936, unit=deg], bmin=[value=0.13008301,
unit=arcsec], bmaj=[value=0.159367442, unit=arcsec]]
```

A novel sort-less gridding algorithm The visibility data are not sorted before the gridding step. Instead, a cache of tiles is allocated to hold each baseline as it moves around in the Fourier plane. When a baseline moves off an existing tile, the results are written to disk and the necessary new tile is read in. Since the rotation of baselines in the uv plane is usually quite slow, the hit rate of such a cache is high. The size of the cache is by default set to half the physical memory of the machine, as specified by the aipsrc variable system.resources.memory. This can be overridden by the user, via the setoptions tool function. The cache can be made smaller at the expense of more paging of tiles in and out. The tile size can also be changed but this is seldom needed. This approach is optimal for arrays with small numbers of antennas but can be slow for *e.g.* the VLA. We intend to rectify this in the near future.

Plug-in commands imager can be customized by attaching commands using the AIPS++ plug-in system. See the file code/trial/apps/imager/imager_standard.gp for an example of how to attach commands.

Suite of tests imager has a suite of tests. A standard test data set and component list can also be created.

imagerwizard The simplemage function is a wizard that performs interactively guided imaging of synthesis data.

dragon The dragon tool performs wide-field imaging using imager.

vpmanager The vpmanager tool manages specification of primary beams for imager.

Near-field imaging *experimental* Images of objects in the near-field of an array can be made. If the distance to the object is specified in setimage, then the extra delay due to the wavefront curvature is corrected in the transforms. Note that some telescopes (*e.g.* VLA) make this correction in the real-time system. This effect is important if the distance to the object is comparable to or less than:

$$\frac{B^2}{\lambda} \tag{1.1}$$

where B is the baseline. Note that the sign of the correction could be in error in this experimental version: try using a negative distance as well as a positive distance.

What imager **needs:**

imager operates on a specified MeasurementSet to produce any of a range of different types of image: dirty, point spread function, clean, residual, *etc.* A MeasurementSet is the holder for measurements from a telescope. It is simply an AIPS++ Table obeying certain conventions as to required and optional contents. The intention is that it should contain all the information needed to reduce synthesis and single dish observations (see AIPS++ Note 191). A UVFITS file can be converted to a MeasurementSet using the fitstoms tool function (a constructor of the ms tool.
imager adds some extra columns to the MeasurementSet to store results of processing. The following columns in the MS are particularly important:

DATA The original observed visibilities are in a column called DATA. These are not altered by any processing in AIPS++.

CORRECTED_DATA During a calibration process, as carried out by *e.g.* calibrater, the visibilities may be corrected for calibration effects. This corrected visibilities are stored in a column CORRECTED_DATA which is created on demand by calibrater and imager. In creating the CORRECTED_DATA column, imager will only correct for parallactic angle rotation. This can be controlled using the correct tool function. All imaging performed by imager is from the CORRECTED_DATA column (apart from the tool function makeimage which can also make dirty images from the other visibility columns).

MODEL_DATA During various phases of processing, the visibilities as predicted from some model are required.

These model visibilities are stored in a column MODEL_DATA. These are used by the calibrater tool for calibration.

IMAGING_WEIGHT Weighting of data (including natural, uniform and Briggs weighting, and tapering) is accomplished by setting the column IMAGING_WEIGHT appropriately.

Standard tools such as the table module and the ms can be used to access and possibly change these (and all other) columns.
imager can handle an initial model in a number of forms: as an image, as a list of images, as a componentmodels:componentlist, or as some combination. Fitting of componentmodels is planned but is not currently supported.
imager uses a number of scratch files. Following AIPS++ practice, these are placed in the directories specified in the aipsrc variable user.directories.work. Those disks that possess sufficient free disk space are chosen in sequence. So to spread your scratch files over two disks each of which has a directory tcornwel/tmp do *e.g.*

```
user.directories.work:   /bigdisk1/tcornwel/tmp /bigdisk2/tcornwel/tmp
```

How to control imager:

To use imager, one has to construct a imager tool using a MeasurementSet as an argument, for example:

```
myimager:=imager('3C273XC1.ms')
```

The Glish variable myimager then contains the tool functions that may be used to do various operations on the MeasurementSet 3C273XC1.ms. These tool functions can be broken down into those that set imager up in some state, and those that actually do some processing. The setup tool functions are:

setimage is a *required* tool function that defines the parameters (size, sampling, phase center, *etc.*) of any image that is to be constructed. If you omit to call setimage prior to any operation that needs these parameters, an error message will result. setimage is passive: nothing happens immediately but subsequent processing is altered.

setdata is an *optional* tool function that selects which data are to be operated on during the processing. This selection can consist of choosing the spectral windows or fields that are to be operated on, or setting channels that are to be operated on in subsequent processing. setdata is active: the selection occurs immediately and is effective for all subsequent operations (until setdata is called again).

setoptions is an *optional* tool function that sets parameters of lesser importance such as gridding parameters, cache sizes. While these affect the processing, usually the default values will suffice. setoptions is passive: nothing happens immediately but subsequent processing is altered.

setbeam is an *optional* tool function that sets the parameters of the synthesized beam to be used in restoring deconvolved images. setbeam is passive: nothing happens immediately but subsequent processing is altered.

setvp is an *optional* tool function that sets the parameters of the voltage pattern model used in mosaicing. setvp is passive: nothing happens immediately but subsequent processing is altered.

setsdoptions is an *optional* tool function that sets the relative scaling and weighting of single dish data versus interferometer data and also other single dish specific parameters like the convolution support when doing single dish imaging.

Thus to understand what imager is doing, one has to remember that at any time, it has a *state* that has been set by using these tool functions. The state may be viewed in one of two ways: either summary can be used to output the current state to the logger, or, in the GUI, the current state of these parameters is displayed and updated following any relevant changes.
All the other tool functions of imager are active: something happens immediately. Hence, for example, the weight tool function acts immediately to change the weighting of the selected data. In particular, unlike other packages, it does *not* set the weighting parameters for latter operations. The clean tool function performs a clean deconvolution of an image, reading and writing a model image. Note that operations that require or produce an image usually take an appropriate image name in the argument list. Often if such an image is not given then it is constructed using the image parameters set via setimage and using an appropriate name (*e.g.* a restored image is named from the model image by appending .restored so that 3C273XC1.clean becomes 3C273XC1.clean.restored).
The concept of the *state* of imager bears a little more explanation. The MeasurementSet can potentially contain data for many different fields and spectral windows. One therefore has to have some way of distinguishing which data are to be included in processing. Rather than have each possible tool function (*e.g.* weight, image, clean) take a long list of parameters to determine

which data are to be included, imager has a setdata tool function that sets imager up some that in subsequent processing only the selected data are processed. For example, to select only field id 1 and spectral windows 1 and 2, one would do:

```
myimager.setdata(fieldid=1, spwid=1:2)
```

The state of imager also consists of information about the default image settings (set via setimage) and various less important options (set via setoptions).

What imager produces:

imager reads and writes AIPS++ MeasurementSets and Images. The format of images is 4 dimensional, with the first two being right ascension and declination, the third being polarization and the fourth being frequency. By suitable choice of the input parameters, one can make images of I alone, I,V or I,Q,U,V for one or all channels. The makeimage tool function can also make a complex image of the original polarizations *e.g.* RR, RL, LR, LL. This latter type of image is useful for diagnostic purposes. Images generated by imager may be viewed using viewer tool or retrieved using the images tool, the MeasurementSets may be accessed using the ms tool. More on this in the example below.

What imager does not do:

imager does not handle calibration of visibility data beyond correction for parallactic angle variations. Instead, you should use the calibrater tool for this purpose. However, imager and calibrater can cooperate on the self-calibration of data.

What improvement to imager are in the works:

We are currently working on a number of improvements:

- Improved gridder to handle many telescopes and many channels more efficiently.
- Parallelized CLEAN and gridding

Advanced use of imager:

As with all AIPS++ applications, imager is designed to be open: all the results are written to and read from standard AIPS++ table files. This open design of imager also allows the user to try out new methods of processing data. Models may be read into Glish, edited or manipulated via standard Glish facilities, and then written out and used subsequently in imager. Suppose that we want to halve the Stokes I of all pixels with negative Stokes I. The following Glish fragment does the trick:

```
m:=image('myimage')
shape:=im.shape()
blc:=[1,1,1,1]
trc:=[shape[1],shape[2],1,shape[4]]
a:=m.getchunk(blc,trc)
a[a<0.]*:=0.5
m.putchunk(a,blc)
m.flush()
m.close()
```

Overview of `imager` **tool functions:**

Data access `open, close, done`

Data selection `setdata`

Data editing `clipvis`

Data calibration `correct`

Data examination `plotvis, plotuv, plotweights, plotsummary`

Weighting `weight, filter, uvrange, sensitivity, fitpsf, plotweights`

Image definition `advise, setimage, make`

Imaging `makeimage, clean, nnls, mem, pixon, restore, residual, approximatepsf, fitpsf, setbeam, ft, smooth feather makemodelfromsd`

Masks `mask, boxmask, regionmask, exprmask, clipimage`

Miscellaneous `summary, setoptions setoptions setoptions`

```
Example
The following example shows the quickest way to make a CLEAN
image and display it.  Note that this can be more easily done
from the toolmanager.
```

```
include 'imager.g'
#
# First make the MS from a FITS file:
#
m:=fitstoms(msfile='3C273XC1.MS', fitsfile='3C273XC1.FITS'); m.close();
```

```
#
# Now make an imager tool for the MS
#
imgr:=imager('3C273XC1.MS')
#
# Set the imager to produce images of cellsize 0.7 and
# 256 by 256 pixels
#
imgr.setimage(nx=256,ny=256, cellx='0.7arcsec',celly='0.7arcsec');
#
# Wait for results before proceeding to the next step
#
dowait:=T
#
# Make and display a clean image
#
imgr.clean(niter=1000, threshold='30mJy',
model='3C273XC1.clean.model', image='3C273XC1.clean.image')
dd.image('3C273XC1.clean.image')
#
# Fourier transform the model
#
imgr.ft(model='3C273XC1.clean.model')
#
# Plot the visibilities
#
imgr.plotvis()
#
# Write out the final MS and close the imager tool
#
imgr.close()
```

Tools

dragon	Tool for wide-field synthesis imaging
imager	tool for synthesis imaging
vpmanager	Tool for specifying voltage patterns and primary beams

Functions

dragontest	Test dragon using a VLA 74 MHz data set
imageralltests	Perform all imager tests using standard MeasurementSets
imagercomponenttest	Perform a test of component model processing using a standard MeasurementSet
imagerlongtest	Perform a long test of imaging from a standard MeasurementSet
imagermaketestcl	Make a standard componentlist for a standard MeasurementSet
imagermaketestmfms	Make a standard multifield MeasurementSet that can be used for experimentation