1.8.12 servers - Tool

Package utility
Module tasking

A tool for sending events between glish and precompiled executables
include "servers.g"

Description
servers communicates between glish distributed objects (tools) and C++ executables. It provides the following services:

- Creation, execution and destruction of C++ servers
- Parameter passing to and from servers
- Plotting, progress meters, *etc.* (implemented in Glish or via a Glish layer)
- Synchronous and asynchronous execution of functions
- Logging of messages

There is only one servers tool running at any one time called defaultservers. This *must* be running if the tool system is to work at all.
Users normally need not interact with defaultservers. Its functions will largely be used by programmers writing tools and most of the interactions are hidden inside the tool framework (this is described in Note 197). The only major exceptions are:

- When one wants to retrieve values returned by a function that was running asynchronously. This is described in the result function.
- When one wants the name of a plotter used by a tool. This is returned by the getplotter function

The terminology used here needs some explanation:

server A server is a C++ executable that runs as a process.

object A server may contain one or more objects. An object is equivalent to a Glish tool. It usually has multiple functions that may be executed.

Some servers are pre-started to minimize AIPS++ startup time. The default pre-started clients are: timer, misc, and quanta. If you wish to prevent this, set the following in your .aipsrc file:

`user.prestart: none`

If you wish to extend or change the list, set the following in your .aipsrc file:

`user.prestart: timer misc quanta measures`

Integers are used to index into lists of servers and objects. Thus to specify a server requires one integer, and to specify an object requires two: one for the server and one for the object. Thus the functions of servers require one or both of these, depending on whether the operation is server or object-specific.

Constructors

Functions

activate	Create a new server
add	Add an object to an existing server
agents	Return a record of all agents
alerter	Relay events on completion
busy	Is the server busy?
create	Create an object
done	Delete a tool
getplotter	Get the plotter used by a tool
info	Print information on the existing servers.
init	Initialize servers
isvalid	Is this a valid server?
kill	Kill a given agent
memory	Return the memory usage of each server
methods	Return the methods of an object
result	Get the result(s) from an asynchronous function.
run	Run a function in an object
running	Determine if an asynchronous function is running
sendmemoryevents	Enable the generation of memory events
settimeout	Set event timeouts
stopmemoryevents	Disable the generation of memory events
suspend	Suspend certain clients
timeouts	Return record of timeouts
trace	Trace events related to certain clients

servers.activate - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Create a new server

Synopsis
activate(server, host, forcenewserver, async, terminateonempty)

Description

The activate function creates a new server of the specified type, if necessary. A server is an executable piece of code that has been written to communicate with glish. This function will start the specified process on the specified computer or, by default, on the current computer ie., the one running glish. This function will only create new server process if one is not already running on the specified computer. However this can be overridden.
This function returns a positive integer that can be used by other functions in this tool indicate which server process you are referring to.
Normally servers are terminated when empty ie., contains no tools. One can override this using the terminateonempty flag. Use this only for servers that are likely to be restarted soon to avoid the overhead for reloading the executable.

Arguments

server	The name of the server to create Allowed:a string Default: no default
host	The name of the computer on which to start the server Allowed:any computer name Default: an empty string
forcenewserver	Start a new server even if one is running Allowed:T or F Default: F
async	Start the server asynchronously Allowed:T or F Default: F
terminateonempty	Terminate servers when empty Allowed:Boolean Default: T

Returns
A positive integer

Example

```
include 'servers.g';
private := [=];
private.serverid := defaultservers.activate('numerics');
private.toolid := defaultservers.create(private.serverid, 'fftserver');
```

This will start up the numerics process on the current machine. It then creates a fftserver tool, using the default constructor, that uses this server. See also the example for the create function.

---

servers.create - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Create an object

Synopsis
create(id, type, creator, invokerecord)

Description

The create function creates an object. The object that is created must be one of those inside the specified server.
Every C++ object that can be created with this function has a className member function. The string returned by this function is the string that must be specified in the type argument to create an instance of that class.
Some objects may be constructed only using a default constructor. For these objects the creator and invokerecord arguments should be left at their default values.
Objects which can be constructed in multiple ways will have a "factory" class associated with them. This factory class will map the string specified in the creator argument into the appropriate constructor.
Some constructors may require additional parameters and these are specified using fields in the invokerecord argument. The parameters that are relevant are defined using Parameter objects in the appropriate factory class.

Arguments

id	Identifies which server to use Allowed: A positive integer obtained from the activate function. Default: no default
type	The object to create. Allowed: A string that corresponds to the output of the className function in the C++ class. Default: no default
creator	A string indicating which constructor to use. Allowed: An empty string or one that matches a string in the "factory" class. Default: An empty string
invokerecord	Additional construction parameters Allowed: A record with fields containing additional construction parameters. Default: An empty record

Returns
A record whose fields?????

Example

```
include 'servers.g';
private := [=];
private.serverid := defaultservers.activate('componentlist');,
private.toolid := defaultservers.create(private.serverid, 'componentlist',
                                        'readcomponentlist',
                                        [filename='crux.cl', readonly=T]);
```

This will start up the componentlist process on the current machine. It then creates a componentlist object, using the "readcomponentlist" constructor. This constructor has two arguments the file to open and whether it should be opened read-only.

---

servers.add - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Add an object to an existing server

Synopsis
add(agentid, id)

Description

The add function adds an object to an existing server. The id that is specified is presumed to have been returned by a function returning a ID.

Arguments

agentid	Identifies which server to use Allowed: A positive integer obtained from the activate function. Default: no default
id	Identifies the object (as returned from a DO) Allowed: A record returned from a DO function Default: no default

Returns
A record

Example

```
id := defaultservers.run(private.serverid, private.invokerec);
toolid := defaultservers.add(private.serverid, id);
return _define_componentlist(private.serverid, toolid);
```

The run returns an id record. The add function is then used to add this id to the list of objects stored in private.serverid. Finally a new componentlist tool is returned using this server and id.

---

servers.run - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Run a function in an object

Synopsis
run(id, invokerecord, async)

Description
The run function runs a member function in the specified C++ object.
The server that will be used is specified by the id parameter. The object, function within that object, and parameters within that function are specified using the invokerecord argument.
There are two special fields in the invokerecord argument. These are "_method" and "_sequence". The first of these indicates which function to run and hence should be a string that matches one returned by the methods member function in the C++ object. I do not know what the "_sequence" argument does but I guess that it indicates which C++ object to use. The value to use for the second argument is the _sequence field in the record returned by the create function as shown in the example below.
Other fields in the invokerecord function are only required if the specified function needs user supplied parameters. The field names must match the name given to the "Parameter" object associated with the specified function.
The async argument controls whether or not to wait for the execution of C++ function to complete. Functions that are quick to compute are normally run synchronously. When a function is run synchrounously the return value of this function is the return value of the C++ function, or T if the C++ function returns a void.
When the function is run asynchronously the return value is a small integer that identifies the job. This job id can be used in conjunction with the running function to determine when the function has finished. When a function has finished the job id can be used in conjunction with the result function to get the return value of the function.
This function can also return a fail if something bad has happened.

Arguments

id	Identifies which server to use Allowed: A positive integer obtained from the activate function. Default: no default
invokerecord	Parameters for the function that will be run Allowed: A record with the required fields Default: no default
async	Return before the function has completed? Allowed: T or F Default: F

Returns

T, an integer or a fail

Example

```
private.getlabelRec := [_method = 'getlabel',
_sequence = private.toolid._sequence];
public.getlabel := function(which) {
  wider private;
  private.getlabelRec.which := which;
  return defaultservers.run(private.serverid, private.getlabelRec);
}
```

This example shows how to use the run function to call a C++ function that has one parameter, the which argument. This function is always run synchronously and hence the return value of the run function is the return value of the C++ function ie., the label string.

---

servers.done - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Delete a tool

Synopsis
done(id, objectid)

Description

The done function closes ie., it calls the destructor, of the C++ object associated with the specified tool. When all the objects that use a server have been shutdown the done function will also shutdown the server.
Once a object has been shutdown it is generally advisable to diasble the tool associated with the C++ object. An example of how this is done is given below.

Arguments

id	Identifies which server to use Allowed: A positive integer obtained from the activate function. Default: no default	
objectid	The object to delete. Allowed: The objectid field of the record returned by the create function. Default: no default	

Returns
T or a fail

Example

```
public.done  := function() {
  wider public, private;
  ok := defaultservers.done(private.serverid, private.toolid.objectid);
  if (ok) {
    val private := F;
    val public := F;
   }
  return ok;
}
```

Assuming the toolid and serverid variables have been defined as shown in the examples for the activate or create functions, it is possible to shutdown the tool and perhaps the server using this function.

---

servers.running - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Determine if an asynchronous function is running

Synopsis
running(jobid)

Description

The running function determines whether the specified job, or asynchronous invokation of a C++ function, has finished. It returns T if the job is still running and F otherwise. If a bad job id is passed it returns a fail.

Arguments

jobid	Which job id we are inquiring about Allowed: A positive integer obtained from the run function. Default: no default

Returns
T, F or a fail

Example
See the example for the results function.

---

servers.result - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Get the result(s) from an asynchronous function.

Synopsis
result(jobid, clear)

Description
The results function is used to get the return values after an asynchronous C++ function has finished.
The results for the specified function are obtained using the jobid that is returned when the function is started with the `run` function. Normally these results are cleared when `result` is called; however, it is possible to keep them for future access by setting the `clear` argument to `F`.
The results are provided in a record format with fields corresponding to each of the input parameters. This is to allow some input parameters to also act as output values. If the C++ function returns a value then the record will also contain a `returnval` field that contains the value returned by the C++ function.

Arguments

jobid	Which job id we are inquiring about Allowed:A positive integer obtained from the run function. Default: no default
clear	Delete the results from the internal buffer Allowed:T or F Default: T

Returns
T, F or a fail

Example

```
private.simulateRec := [_method = 'simulate',
     _sequence = private.id._sequence];
public.simulate := function(howmany=1, async=F) {
  wider private;
  private.simulateRec.howmany := howmany;
  retval := defaultservers.run(private.serverid,
                             private.simulateRec, async);
  if (is_fail(retval)) return retval;
  if (async) {
    t := client("timer", 1.0);
    whenever t->ready do {
   if (!defaultservers.running(retval)) {
        t->terminate();
     deactivate;
     note('Added ', howmany, ' simulated component(s) to the list.',
          origin='componentlist.simulate');
   }
    }
  } else {
    note('Added ', howmany, ' simulated component(s) to the list.',
      origin='componentlist.simulate');
  }
  return T;
}
```

This example illustrates how a function can be made to work both synchronously or asynchronously. In either the former case the user waits for the function to complete does not receive a glish prompt until then. In the latter case the functions returns quickly and executation proceeeds in the backgroud. In either case the note is only printed when everything has been done.

---

servers.getplotter - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Get the plotter used by a tool

Synopsis
getplotter(id, name)

Description
When a tool uses a plot command, a pgplotter is created. The plotter is stored by `defaultservers`, and a reference to it may be retrieved using the getplotter function. Since it is a glish reference to the `pgplotter`, commands that change the state of the plotter may be issued. The main use fot this is to allow script to issue print commands to the pgplotter.
The function needs the agentid field in the record returned from the `id()` function of the tool. All tools should have this function, but unfortunately some may not.

Arguments

id	agentid field from record returned by tool id() function Allowed:A positive integer Default: no default
name	Optional name of plotfile Allowed:String Default: ”

Returns
A record containing pgplotters or a fail

Example

```
myimager := imager('3C273XC1.ms');
myimager.plotuv();
imagerplotter := defaultservers.getplotter(myimager.id().agentid)
imagerplotter[1].postscript('3C273XC1.uvplot.ps');
      }
```

This example illustrates how to retrieve and use the `pgplotter`. Note that We assume in this example that the only plotter is the first one. In more complex cases, one might have to use a different element in the returned record.

---

servers.isvalid - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Is this a valid server?

Synopsis
isvalid(agentid)

Arguments

agentid	server id Allowed:Int Default: no default

Returns
Bool

---

servers.busy - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Is the server busy?

Synopsis

busy(agentid)

Arguments

agentid	server id Allowed:Int Default: no default

Returns

Bool

---

servers.settimeout - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Set event timeouts

Synopsis

settimeout(action, timeout)

Description

Normally some crucial events are just awaited. It is possible to set a timeout for some of the events. Normally this should not be needed except in debugging. The events are:

create Create a server

close Close a tool

done Close a server

makeplot Make a plot server

Arguments

action	event name Allowed:String Default: create
timeout	Time out in seconds Allowed:Numeric Default: 60

Returns
Bool

---

servers.timeouts - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Return record of timeouts

Synopsis
timeouts()

Description
Normally some crucial events are just awaited. It is possible to set a timeout for some of the events. Normally this should not be needed except in debugging. The events are:

create Create a server

close Close a tool

done Close a server

makeplot Make a plot server

Returns
Record of timeouts

---

servers.info - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Print information on the existing servers.

Synopsis
info()

Returns
Bool

---

servers.memory - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Return the memory usage of each server

Synopsis
memory()

Description
servers keeps track of how much memory each server is using. This function will return a record containing the amount of memory, in mega-bytes, used bt each server. For convienience the amount of memory used by Glish is also returned.

Returns
record

Example

```
include 'servers.g';
print defaultservers.memory();
# [misc=0.290412903, quanta=0.0799331665, Glish=3.77894211]
```

---

servers.sendmemoryevents - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Enable the generation of memory events

Synopsis
sendmemoryevents()

Description
servers can send a memory event whenever it detects that the memory used by a server has changed. The value associated with the memory event is a record identical to the one returned by the memory function.
By default servers does not send events and this function must be called to initiate this behaviour. The complimentary function is stopmemoryevents.

Returns
nothing

Example

```
include 'servers.g';
whenever defaultservers->memory do {
  local memory := $value;
  total := 0;
  for (i in ind(memory)) {
     total +:= memory[i];
  }
  print 'aips++ memory usage is', total;
}
defaultservers.sendmemoryevents();
# At this point the whenever may receive events
defaultservers.stopmemoryevents();
# At this point the whenever will not receive any events.
```

This example creates a "whenever" statement that will print the total memory used by aips++. However the defaultsaervers will not emit memory events until the sendmemoryevents function is called.

---

servers.stopmemoryevents - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Disable the generation of memory events

Synopsis
stopmemoryevents()

Description
servers can send a memory event whenever it detects that the memory used by a server has changed. This function can be used to prevent this from happening. It is complementry to the sendmemoryevents function.
Using this function does does not guarentee that servers will stop sending memory events. servers counts how many times the sendmemoryevents function has been called and the stopmemoryevents function must be called an equal number of times to actually stop the memory events. This is to ensure that one client does not stop events for other clients that have requested them.

Returns
nothing

Example
See the example for the sendmemoryevents function.

---

servers.init - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Initialize servers

Synopsis
init()

Description
This function initializes servers after construction. It should not be called other than in servers.g.

Returns
don't know

---

servers.alerter - Function

Package **utility**
Module **tasking**
Tool **servers**

Relay events on completion

Synopsis
alerter()

Description
servers works internally by sending and receiving events. When running synchronously these events are converted into return values as appropriate. However, when running asynchronously, one needs a mechanism to emit events that can be caught outside. The alerter function provides this mechanism. The events emitted are:

error The job ended in error.

fail The server failed.

note A note was issued.

'jobid' The job with id 'jobid' ended successfully.

The value returned by the event is a record with fields:

agent The agent used in execution.

value The value.

Here is an example of how to execute a function asynchronously and set up a whenever on the result:

```
im:=imager('3C273XC1.ms')
jobid:=im.plotvis(async=T);
whenever defaultservers.alerter()->[as_string(jobid)] do {
  print 'Plot finished';
  deactivate;
}
```

Returns
Agent record for receiving events and their results

---

servers.agents - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Return a record of all agents

Synopsis
agents()

Description
Needed for debugging only.

Returns
Record

---

servers.suspend - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Suspend certain clients

Synopsis
suspend(suspend_clients)

Description
Needed for debugging only.

Arguments

suspend_clients	Clients to suspend Allowed:String Default: no default

Returns
Bool

servers.trace - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Trace events related to certain clients

Synopsis
trace(trace_clients)

Description
Needed for debugging only.

Arguments

trace_clients	Clients to trace Allowed:String Default: no default

Returns
Bool

servers.kill - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Kill a given agent

Synopsis
kill(agentid)

Description
Kill a given agent.

Arguments

agentid		Agent id Allowed:Int Default: no default

Returns
Bool

---

servers.methods - Function

Package	**utility**
Module	**tasking**
Tool	**servers**

Return the methods of an object

Synopsis
methods(agentid, objectid)

Description
Return the methods of a given object.

Arguments

agentid		Agent id Allowed:Int Default: no default
objectid		Object id Allowed:Int Default: no default

Returns
String

1.9 benchmark - Module

Module for standard AIPS++ performance benchmarks
include "benchmark.g"

Description

`benchmark` provides a standard set of AIPS++ performance benchmarks.

Performance benchmarks

This module defines the standard AIPS++ performance benchmarks. The primary purpose of the `benchmark` tool is to allow routine and automated performance monitoring of representative data reduction threads. This, in turn, allows performance problems due to unrelated code evolution to be identified and corrected early in development.
Performance benchmarks are invariably an imperfect measure of the performance of any software package. They are inherently difficult to define for a number of reasons, including:

Coverage Benchmarks need to cover a set of operations which together are representative of the average use profile of the system. The heterogeneous nature of large radio astronomy reduction packages, including AIPS++, makes this very difficult to define in any unique or absolute sense. This is further compounded by the broad range of applications for the package, including near real-time use at telescopes and use in off-line reduction environments. The benchmarks defined here attempt to address this balance by focusing on the common scientific reduction threads most frequently in use.

Scientific relevance Benchmarks need to be cast in terms of their scientific relevance if they are to be useful. In a pipeline environment, for example, the reduction time of any given thread is most sensibly expressed as the fraction of the observing time of the data under reduction. In an interactive environment, the relevant time-scale is the time it takes to set input parameters for the operation being performed. In either case, there is a point at which performance improvements no longer improve scientific throughput in any meaningful sense. Conversely, expressing benchmarks in terms of their scientific relevance isolates those cases which should be and can be profitably improved.

Dependence Benchmarks typically depend on a large number of parameters, including uv-data set size, image size and deconvolution algorithm type, amongst many others. Scientific throughput is therefore a scalar function over many dimensions. In addition, the constituent components in the overall performance function are often non-linear in any individual parameter (e.g. $O^n(p)$), so the overall performance function cannot be linearly extrapolated from one part of parameter space to another. Any performance benchmark is only sampling the overall performance function at one point in the parameter hyperspace. This reinforces the need for benchmarks to be representative of a broad range of common data reduction threads.

Environment Benchmark measurements may be very sensitive to the immediate configuration of the local computing environment. This includes factors such as disk I/O protocols (SCSI versus IDE), background operating system I/O caching policies, operating system and architecture, networked (NFS) or non-networked disk access, available memory and memory utilization policies in the applications. All influence the measurement obtained for any given benchmark, and need to be carefully controlled for reliable benchmark measurements.

Inter-comparisons Performance comparisons between different software systems are particularly difficult, primarily because it is seldom possible to inter-compare exactly the same operation in each package, due to differences in implementation and approach. In addition, great care needs to be taken in controlling the environment in making such comparisons. Packages which have small memory models are difficult to compare with packages which have large memory models, for example, given the substantial difference in access times for main memory and disk. The benchmarks defined here do however include scripts for comparable (or similar) operations in other packages in order to provide an approximate estimate of relative performance. However such numbers are invariably highly unreliable for the reasons stated here.

Overview of `benchmark` tool functions:

This section contains a list of the `benchmark` tool functions by category.

All benchmarks all

Clark CLEAN single-field imaging benchmarks ccsf ccsfvl4uc

Example
The following example shows how to run all benchmarks.

```
include 'benchmark.g'
#
# Create a benchmark tool
#
bench:=benchmark();
#
# Run all defined benchmarks
#
bench.all();
#
# Close the benchmark tool
#
bench.done()
```

Description of individual benchmarks

Each benchmark is identified by a unique benchmark code. These are hierarchical, and specify all relevant parameters applicable to the benchmark. This section includes any scripts from other packages which perform the same reduction. Package inter-comparison is subject to the caveats mentioned above however.

Clark Clean single-field imaging

This benchmark has the following code descriptor:

$$\text{CC-SF-}data\text{−SP}win - stokes - weight - npixel - nclean$$

where *data* is the input data description, further broken down as *data*=[*instrument*][*size*][*compression*][*obsmode*]. The parameters are specified as follows: i) *instrument* - telescope abbreviation (VL=VLA); ii) *size* - dataset size on an integer scale of 1 to 10 (4 ∼ 630k UVFITS visibilities); iii) *compression* - MS compression (U=uncompressed; C=compressed); iv) *obsmode* - observing mode (C=continuum, L=line); v) *win* - number of spectral windows imaged; vi) *stokes* - Stokes parameters imaged (e.g. IQUV or I); vii) *weight* - imaging weight (UN=uniform, NA=natural); viii) *npixel* - image size in pixels; vix) *nclean* - number of Clean components in deconvolution.
The individual Clark Clean single-field benchmarks implemented in the default run of the *ccsf*() method at present are enumerated in the tables below. The common portion of benchmark code mentioned in the table caption. The variable part of the code is listed in the "Benchmark" column of the tables. Full benchmark code can be constructed by concatenating the constant and the variable part of the code

The corresponding AIPS script for these benchmarks is as follows. The environment variable `DATA` must be set to the point to the directory where the benchmark related data files are kept (`/aips++/data/demo/benchmark` by default). Note that since AIPS++ datafile names have lower case letters, you will have either make a symbolic link (speaking for UNIX only) or copy the relevant data files to a files with all capital letters. This is required since AIPS internally converts all letters to upper case.
`<FITSFILE>` in the following script is the name of the UV FITS file to be loaded (E.g. VLAC125K.FITS).
`<N>` is the AIPS catalog number corresponding to the UVDATA file loaded using `FITLD`.
`<NPIX>` is the size of the image in pixels.
`<STOKES>` is 'I', 'Q', 'U', or 'V'. `IMAGR` task in AIPS accepts a single stokes value per execution.
`<WEIGHT>` is 'U' for uniform weighting or 'N' for natural weighting.
`<NITER>` is the number of Clean iterations required.

```
TASK 'FITLD'
```

Table 1.862: Table for continuum VLA 125K rows dataset benchmark. Common part of the benchmark code: 'CC-SF-VLAC-U125K-SP1-'

Benchmark code	Dataset	N_{spw}	Stokes	Wgt.	N_{pixel}	N_{chan}	N_{clean}
I-UN-512-C1-1000	vlac125K	1	I	UN	512	1	1000
IQUV-UN-C1-512-1000	vlac125K	1	IQUV	UN	512	1	1000
I-UN-1024-C1-1000	vlac125K	1	I	UN	1024	1	1000
IQUV-UN-1024-C1-1000	vlac125K	1	IQUV	UN	1024	1	1000
I-UN-2048-C1-1000	vlac125K	1	I	UN	2048	1	1000
IQUV-UN-2048-C1-1000	vlac125K	1	IQUV	UN	2048	1	1000
I-NA-512-C1-1000	vlac125K	1	I	NA	512	1	1000
IQUV-NA-512-C1-1000	vlac125K	1	IQUV	NA	512	1	1000
I-NA-1024-C1-1000	vlac125K	1	I	NA	1024	1	1000
IQUV-NA-1024-C1-1000	vlac125K	1	IQUV	NA	1024	1	1000
I-NA-2048-C1-1000	vlac125K	1	I	NA	2048	1	1000
IQUV-NA-2048-C1-1000	vlac125K	1	IQUV	NA	2048	1	1000

Table 1.863: Table for continuum VLA 1M rows dataset benchmark. Common part of the benchmark code: 'CC-SF-VLAC-U1M-SP1-'

Benchmark code	Dataset	N_{spw}	Stokes	Wgt.	N_{pixel}	N_{chan}	N_{clean}
I-UN-512-C1-1000	vlac1M	1	I	UN	512	1	1000
IQUV-UN-512-C1-1000	vlac1M	1	IQUV	UN	512	1	1000
I-UN-1024-C1-1000	vlac1M	1	I	UN	1024	1	1000
IQUV-UN-1024-C1-1000	vlac1M	1	IQUV	UN	1024	1	1000
I-UN-2048-C1-1000	vlac1M	1	I	UN	2048	1	1000
IQUV-UN-2048-C1-1000	vlac1M	1	IQUV	UN	2048	1	1000
I-NA-512-C1-1000	vlac1M	1	I	NA	512	1	1000
IQUV-NA-512-C1-1000	vlac1M	1	IQUV	NA	512	1	1000
I-NA-1024-C1-1000	vlac1M	1	I	NA	1024	1	1000
IQUV-NA-1024-C1-1000	vlac1M	1	IQUV	NA	1024	1	1000
I-NA-2048-C1-1000	vlac1M	1	I	NA	2048	1	1000
IQUV-NA-2048-C1-1000	vlac1M	1	IQUV	NA	2048	1	1000

Table 1.864: Table for spectral line VLA 125K rows, 64 channel dataset benchmark. Common part of the benchmark code: 'CC-SF-VLAL-U125K-SP1-'

Benchmark code	Dataset	N_{spw}	Stokes	Wgt.	N_{pixel}	N_{chan}	N_{clean}
I-UN-512-C64-1000	vlac125K	1	I	UN	512	64	1000
I-UN-1024-C64-1000	vlac125K	1	I	UN	1024	64	1000
I-UN-2048-C64-1000	vlac125K	1	I	UN	2048	64	1000
I-NA-512-C64-1000	vlac125K	1	I	NA	512	64	1000
I-NA-1024-C64-1000	vlac125K	1	I	NA	1024	64	1000
I-NA-2048-C64-1000	vlac125K	1	I	NA	2048	64	1000

Table 1.865: Table for continuum BIMA 139M rows dataset benchmark. Common part of the benchmark code: 'CC-SF-BIMAL-U139M-SP1-'

Benchmark code	Dataset	N_{spw}	Stokes	Wgt.	N_{pixel}	N_{chan}	N_{clean}
I-NA-256-C20-1000	BIMADATA	1	I	NA	256	20	1000

```
INFILE 'DATA:<FITSFILE>'

GO FITLD

TASK 'IMAGR'
GETN         <N>
INSEQ         1
INDISK        1
SOURCES      ' '
QUAL         -1
CALCODE      ' '
TIMERANG      0
SELBAND      -1
SELFREQ      -1
FREQID        1
SUBARRAY      0
DOCALIB      -1
GAINUSE       0
DOPOL        -1
BLVER        -1
FLAGVER       0
DOBAND       -1
BPVER        -1
SMOOTH        0
STOKES     <STOKES>
```

```
BCHAN         1
ECHAN         1
CHANNEL       0
NPOINTS       1
CHINC         1
BIF           1
EIF           1
OUTNAME      ' '
OUTDISK       1
OUTSEQ        0
OUTVER        0
IN2NAME      ' '
IN2CLASS     ' '
IN2SEQ        0
IN2DISK       0
CELLSIZE      5,5
IMSIZE     <NPIX>, <NPIX>
NFIELD        1
DO3DIMAG     -1
FLDSIZE       0
RASHIFT       0
DECSHIFT      0
UVTAPER       0
UVRANGE       0
GUARD         0
ROTATE        0
ZEROSP        0
UVWTFN     <WEIGHT>
UVSIZE        0
ROBUST        0
UVBOX         0
UVBXFN        1
XTYPE         5
YTYPE         5
XPARM         0
YPARM         0
NITER      <NITER>
BCOMP         0
ALLOKAY       0
NBOXES        0
CLBOX         0
BOXFILE      ' '
OBOXFILE     ' '
GAIN          0.1
FLUX          0
MINPATCH     51
```

Table 1.866: Table for VLA calibrator benchmark (for 'G' and 'D' Jones).

Benchmark code	Data	Compressed?	Jones	NAnt	SNR	NSolInt
CALVLAU-G-27-10-100	calvlac27s10.fits	Nope	G	27	10	100
CALVLAU-D-27-10-100	calvlac27s10.fits	Nope	D	27	10	100

```
BMAJ          0
BMIN          0
BPA           0
OVERLAP       0
PHAT          0
FACTOR        0
CMETHOD      ' '
IMAGRPRM      0
NGAUSS        0
WGAUSS        0
FGAUSS        0
MAXPIXEL      0
DOTV         -1

GO IMAGR
```

The equivalent script for spectral line benchmarks would set the keywords `BCHAN=C0`, `ECHAN=C1` where `C0` and `C1` are the start and end frequency channel numbers. This will generate a spectral cube. For multi-frequency synthesis, `NPOINTS` must be set to a value `>=C1-C0+1` (this will grid all the specified channels on a single uv-grid).

VLA G and D term calibration

This benchmark has the following code descriptor:

CALVLA*COMPRESSION*-*JONES*-*NAnt*-*SNR*-*NSolInt*

where *COMPRESSION* is the compression state of the input data (compressed or uncompressed), *JONES* is the name of the Jones matrix being solved for ('G', 'D', or 'B'), *NAnt* is the total number of antennas in the data, *SNR* the signal-to-noise ratio of the input data, and *NSolInt* is the number of solution intervals involved.
The individual calibrator benchmarks implemented in the default run of the *calvla*() method at present are enumerated in the Table 1.866.

The corresponding AIPS script for these benchmarks is as follows. The environment variable `DATA` must be set to the point to the directory where the benchmark related data files are kept (`/aips++/data/demo/benchmark` by default). Note that since AIPS++ datafile names have lower case letters, you

will have either make a symbolic link (speaking for UNIX only) or copy the relevant data files to a files with all capital letters. This is required since AIPS internally converts all letters to upper case.
`<FITSFILE>` in the following script is the name of the UV FITS file to be loaded (E.g. CALVLAU27S10.FITS).
`<N>` is the AIPS catalog number corresponding to the UVDATA file loaded using `FITLD`.
Following is the script for solve for the antenna based complex gain terms (GJones) as a function of time.

```
TASK 'FITLD'
INFILE 'DATA:<FITSFILE>'

GO FITLD
```

This must be followed by a few, simple operations to get the UV database inside AIPS in proper order. The AIPS++ generated FITS files, though are in Time-baseline order naturally, don't seem to indicate so via whatever keyword AIPS looks for. Hence, one needs to run the AIPS task `UVSRT` to write out a Time-baseline (`TB`) sorted UV database as follows:

```
TASK 'UVSRT'
GETN        <N>
SORT        'TB'
GO
```

Next, since the simulated database is a single source database, it does not have the equivalent of the AIPS default `CL` table. This is required by `PCAL` later. To generate the blank `CL` table, one needs to convert the database into a multi-source database (even though it has only one source in it). This is done by running the AIPS task `MULTI`.

```
TASK 'MULTI'

GETN        <N>
GO
```

Next, run the AIPS task `INDXR` as follows:

```
TASK 'INDXR'

GETN        <N>
INFILE      ' '
PRTLEV       0
CPARM        0
BPARM        0
IN2FILE    ' '
GO
```

Finally, run the task SETJY to put the source full stokes flux density in the AIPS SU tables.

```
TASK 'INDXR'

GETN        <N>
SOURCES     ' '
QUAL         -1
BIF           0
EIF           0
ZEROSP       14.76,0.66,1.48,0
OPTYPE      ' '
CALCODE     ' '
SYSVEL        0
RESTFREQ      0 0
VELTYP      ' '
VELDEF      ' '
FREQID       -1
APARM         0
GO
```

The data is now ready to run the tasks CALIB (for the equivalent of GJones calibration) and PCAL (for the equivalent of DJones calibration).

```
TASK 'CALIB'
GETN         <N>
CALSOUR       ' '
QUAL         -1
CALCODE       '    '
SELBAND       0
SELFREQ       0
FREQID        0
TIMERANG      0
BCHAN         64
ECHAN         64
ANTENNAS      0
DOFIT         0
ANTUSE        0
SUBARRAY      0
UVRANGE       0  0
WTUV          0
DOCALIB      -1
GAINUSE       0
FLAGVER       0
DOBAND       -1
BPVER        -1
SMOOTH        0
```

```
IN2NAME      '     '
IN2CLASS     '     '
IN2SEQ        0
IN2DISK       0
INVERS        0
NCOMP         0
FLUX          0
NMAPS         0
CMETHOD      '    '
CMODEL       '    '
SMODEL       0
OUTNAME      '    '
OUTCLASS     '    '
OUTSEQ        0
OUTDISK       1
REFANT        0
SOLINT        1.16 #to get 100 solution intervals for the checked-in benchmark dataset
APARM         0
SOLTYPE      '    '
SOLMODE      'A&P'
SOLCON        0
MINAMPER      0
MINPHSER      0
CPARM         0
SNVER         0
ANTWT         0
GAINERR       0
BADDISK       0

GO CALIB
```

Following is the script for solve for the antenna based polarization leakage term.

```
TASK 'PCAL'
GETN         <N>
CALSOUR      ' '
TIMERANG      0
SELBAND      -1
SELFREQ      -1
FREQID       -1
BIF           0
EIF           0
ANTENNAS      0
UVRANGE       0  0
SUBARRAY      0
```