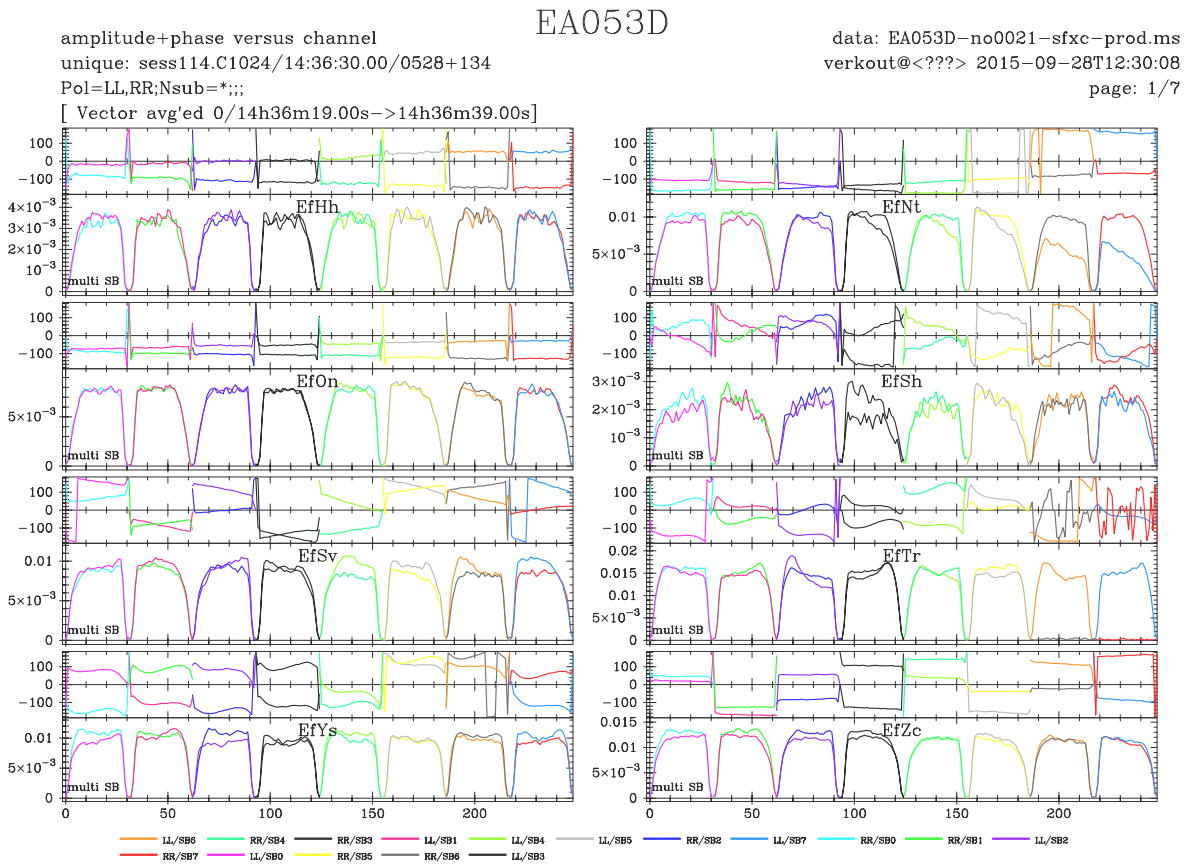
JOINT INSTITUTE FOR VLBI IN EUROPE

OUDE HOOGEVEENSEDIJK 4, DWINGELOO

Telephone: +31 (0) 592 596500
Fax: +31 (0) 592 596539

26 September 2015

TO: Distribution
FROM: Harro Verkouter (verkouter@jive.eu)
SUBJECT: cookbook ***draft v2*** for *jplotter* v0.4 and higher



1. The *jplotter* program

The *jplotter* program is a Python based program for displaying fundamental properties of (time/frequency averaged) radio-astronomical data stored in Measurement Sets (MS)[1], e.g. amplitude and/or phase of the complex data as function of time, frequency or UV-distance, as witnessed above.

This document aims to describe the high-level ideas and features of the program and installation instructions for the dependencies rather than just documenting the commands; specifically because the program comes with all command documentation built-in and available at runtime.

[1] The Measurement Set is both a data-model description as well as an implementation of this model in C++ in the Common Astronomy Software Applications package (CASA) - `http://casa.nrao.edu/Memos/229.html`

2. Running the *jplotter* program

Assuming the jplotter program has been installed (see Appendix A) it can be run:

```
$> /path/to/jplotter [options]
```

Currently the supported command-line options are:

`-h`	help on startup parameters; other options are ignored if –h is present
`-v`	display the current version and exit succesfully
`-d`	run in debug mode: print a stack trace in case of error

The program will display a prompt, at which three useful basic commands are immediately available: **`exit`**, **`list`** and **`help`**:

```
++++++++++++++++++++ Welcome to cli ++++++++++++++++++
$Id: command.py,v 1.13 2015-09-21 11:36:22 jive_cc Exp $
  'exit' exits, 'list' lists, 'help' helps
jcli>
```

From here on, text **`in this font`** will represent a command that can be typed at this prompt. Help on such a command can be retrieved by typing **`help <command>`** at said prompt. In general, *jplotter* commands are one- or two letter long mnemonic abbreviations. Running a command without argument(s) usually displays its current setting or value. *jplotter* features *tab*-filename completion and command line history if readline is installed.

3. Theory of operation

For most plots the basic unit of data is the complex numbers found in the data column of a Measurement Set - the visibility data matrices of dimension n_{freq} x $n_{polarizations}$. Most plot types will display (meta)data of these complex numbers.

In general, plotting using *jplotter* will follow these steps:

1. open a **M**easurement **S**et using **`ms`**
2. choose the **p**lot **t**ype using **`pt`** (**`lp`** to **l**ist the available **p**lot types)
3. [narrow down the data to be displayed - see section **6**]
4. [fiddle with the details of how the plots are presented - see sections **9, 10**]
5. use **`pl`** to tell *jplotter* to (re)**pl**ot the data; navigate multiple pages (section **11**)

jplotter has no particular feelings as to the order in which these steps are done other than the logical ones: it is impossible to (de)select data or make the plots unless a data set is opened to (de)select data or make plots from. Steps in square brackets are optional.

A word of caution about step 3. This one is indicated as an optional step. By default *jplotter* will plot **every complex number** in the Measurement Set, unless it's been told to limit the selection to a particular subset of data or some averaging is to be done! Needless to say that in a typical observation there are quite a few complex numbers[2].

In reality there could be a number of optional sub-steps inserted between opening of a data set and the "go!" signal - it all depends how far down the rabbit hole you want to go.

[2] In theory plotting a large selection should finish, eventually. The information content on screen may be under- (or even over-)whelming though.

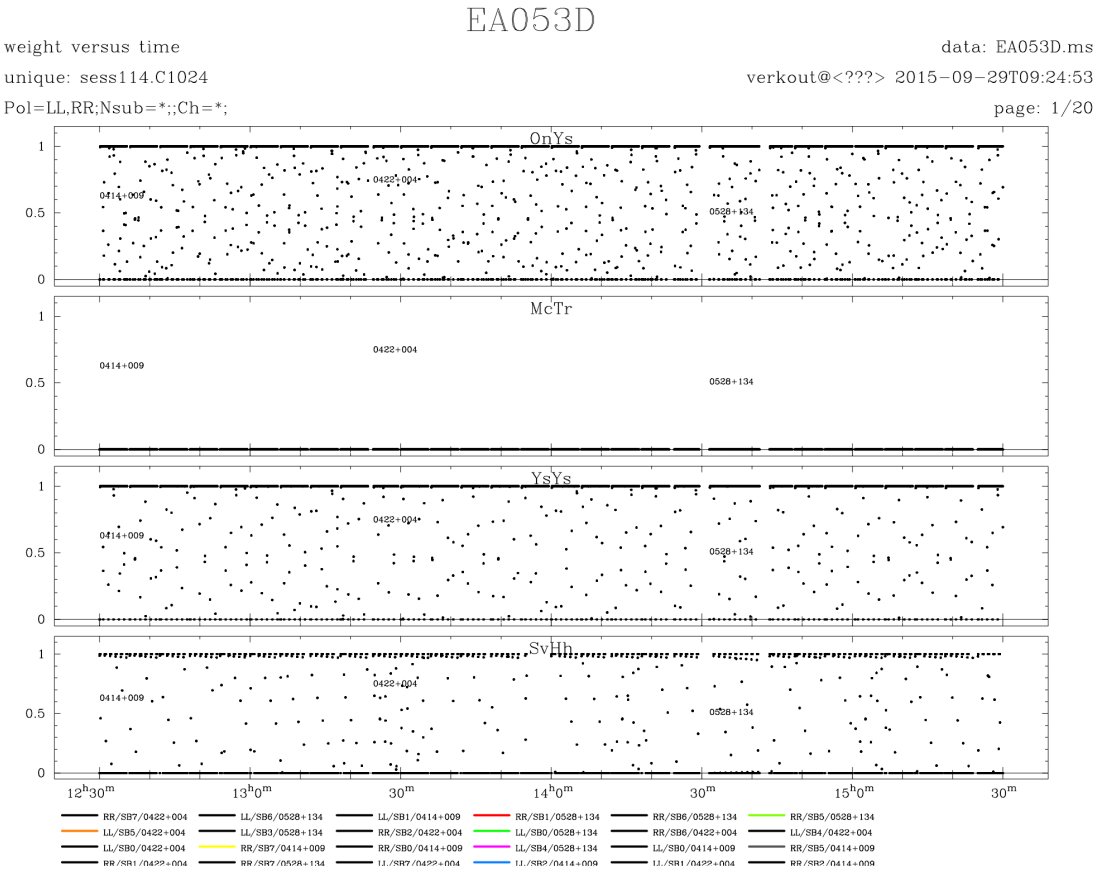
4. The first plots

Let's look at what's possibly the simplest plot(s) that can be made. This is a transcript of a minimal *jplotter* session. The commands to be typed are highlighted.

```
jcli> ms EA053D.ms
ms: Current MS is 'EA053D.ms' containing 5891040 rows of Spectral data for EA053D
jcli> pt wt
plotType:                  wt [weight versus time]
jcli> pl
Data munching took       90.282s
min/max processing took    0.338s
jcli>
```

In words this reads: "Open the EA053D.ms Measurement Set, set plot type to weight-versus-time and plot me this".

Steps **3.** and **4.** of the workflow (data selection, plot presentation; see previous section) have been skipped: it takes ~90 seconds[3] to process all of the approximately 6 million rows of weight data. 20 pages of plots with possibly a little bit too much information to comprehend are created:



The take-away messages here are:

- *nothing* needed to be known about what is contained in the MS; the command was to create weight-versus-time plots and that's what it did. It finds polarizations, baselines, sources &cet. all by itself
- if no data is selected, *everything* is plotted
- each plot type has a default layout and plots are organized by baseline by default

Let's see if we can improve on both the speed and information content!

[3] Plotting can always be interrupted by pressing **^C** (control-C)

The weights on cross-correlations are a combination of the auto-correlation weights for the data that went into that cross-correlation. Therefore all cross-correlation weights provide redundant information.

For useful weight-versus-time plots it is therefore sufficient to find all the auto-correlations and plot their weight.

The baseline selection command **`bl`** can be used to select only the auto baselines. The pseudo baseline name **auto** narrows the data selection to those baselines where both inputs to the baseline are the same antenna:

```
jcli> bl auto
baselines: JbJb WbWb EfEf McMc NtNt OnOn ShSh TrTr YsYs SvSv ZcZc HhHh
```

The software knows by itself which auto-correlation baselines are present in the MS.

Although this is an improvement, it does not strictly select auto-correlations. An auto-correlation is the same signal with itself. If all polarization products are formed by the correlator then this still does not de-select the cross-polarizations.

Through the **`fq`** command it is possible to quickly select only the parallel hand polarizations ("**/p**" for parallel hands; "**/x**" for cross hands) for all subbands ("*****") in the dataset:

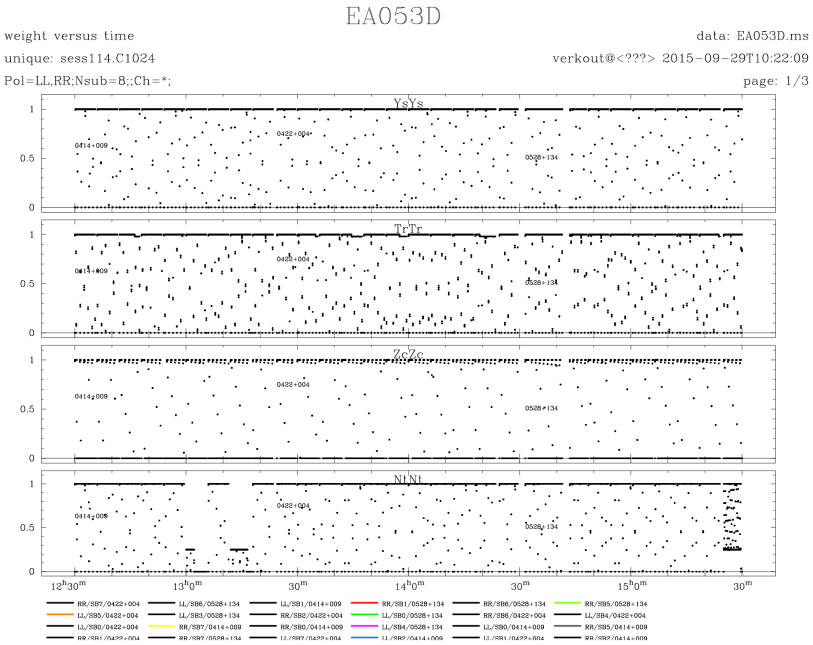
```
jcli> fq */p
freqsel:   0/0:7/0:RR,LL
```

The reply says that from frequency group 0, the polarization combinations RR and LL from subbands 0..7 have been selected - exactly what was desired. Note that this will typically *not* work on ALMA MSs because of their non-standard SPECTRAL_WINDOW table. More on this later.

Re-running the **`pl`**ot command to update the plots yields:

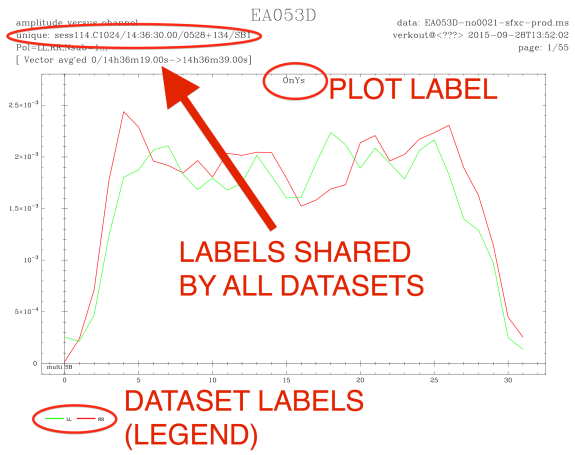
```
jcli> pl
Query took           0.775s
Data munching took       15.498s
min/max processing took    0.049s
```

So now it finishes in 15 seconds and produces only three pages of plots:



5. What's on screen

This is an appropriate time to digress a little. In order to understand how *jplotter* divides data sets into plots, a little background about how data is processed, accumulated and labelled is needed.



jplotter accumulates individual data points into datasets: a series of *(x, y)* points. Each dataset gets values for each of the seven labels: TIME, SRC, BL, FQ, SB, P, CH (see below), bar the current *x*-axis.

All dataset labels are analyzed for common (i.e. redundant) values, which are removed and displayed in the top-left-hand side of the page.

Finally, the distinct combinations of label values for the set of labels set in the **`new`** command define the plots in which a dataset is drawn.

The remaining labels are the actually useful dataset labels and are plotted as legend in the bottom part of the page. The default **`new`** setting is **FQ BL**, saying as much that for each observing mode (**FQ**) and baseline (**BL**) a plot is generated, with all data sets for that combination overplotted.

6. Oh my label!

It may be less obvious from the previous section that the dataset labelling plays an important role in the *jplotter* program. The reality is that the labelling comes up in almost every facet of the plotting from changing the appearance of the plots through data selection.

Throughout the *jplotter* program the following seven label types/names will be almost consistently used:

Label	Is short for	Description
TIME	TIME	The time stamp of the data point
SRC	SOURCE	The name of the associated FIELD of this data point
BL	BASELINE	The baseline on which this data point was measured
FQ	FREQUENCY GROUP	The frequency group name (a.k.a. the frequency setup name or observing mode)
SB	SUBBAND	The subband number within the frequency setup
P	POLARIZATION	The (human readable) polarization combination of the data point
CH	CHANNEL	The channel number of the data point

7. Data selection and labels

jplotter views a Measurement Set as a seven-dimensional array, or rather, the data as having seven axes. It should not come as a surprise that these correspond to the label types mentioned in the previous section.

In order to allow selection of data (the optional 3rd step in the workflow), *jplotter* holds the view that each of the seven axes has a *range* of values which it represents. See the one-letter **`r`** command, which is (very) short for ***r****ange*.

After opening a Measurement Set, the **`r`** command can be used to get a quick overview of what is actually contained in this data set:

```
jcli> ms EA053D.ms
ms: Current MS is 'EA053D.ms' containing 5891040 rows of Spectral data for EA053D
jcli> r
listFreqs:     FREQID=0 [sess114.C1024]
listFreqs:       SB 0: 4926.9900MHz/16.0MHz  32ch P0=RR,LL
listFreqs:       SB 1: 4942.4900MHz/16.0MHz  32ch P0=RR,LL
listFreqs:       SB 2: 4958.9900MHz/16.0MHz  32ch P0=RR,LL
listFreqs:       SB 3: 4974.4900MHz/16.0MHz  32ch P0=RR,LL
listFreqs:       SB 4: 4990.9900MHz/16.0MHz  32ch P0=RR,LL
listFreqs:       SB 5: 5006.4900MHz/16.0MHz  32ch P0=RR,LL
listFreqs:       SB 6: 5022.9900MHz/16.0MHz  32ch P0=RR,LL
listFreqs:       SB 7: 5038.4900MHz/16.0MHz  32ch P0=RR,LL
listAntennas:  Jb ( 0)  Wb ( 1)  Ef ( 2)  Mc ( 3)  Nt ( 4)  On ( 5)  Sh ( 6)
listAntennas:  Tr ( 8)  Ys ( 9)  Sv (10)  Zc (11)  Hh (13)
listSources:   0414+009 0422+004 0528+134
listTimeRange: 06-Mar-2014/12:30:00.500 -> 06-Mar-2014/15:29:59.481  dT: 1.000
```

Depending on the type of the label the selectable range can be either:

list of discrete values	e.g. sources (**SRC**), baselines (**BL**), polarization (**P**)
continuous range	e.g. **TIME** (start-time through end-time)
index array *0..n*	e.g. subbands (**SB**), channels (**CH**), frequency group/setup

Note that the **`r`** command does not, by default, display all baselines but rather just the antennae that were found. The actual list of baselines can be inquired using **`r bl`** ("display the range of the **bl** label").

Through the use of commands like **`time`**, **`src`**, **`fq`**, **`bl`** or **`ch`** the data selection can be narrowed to the desired subset of values. The **`sl`** command - from **s**el**l**ection - gives an overview of the full current selection:

```
jcli> sl
freqsel:   0/0:7/0:RR,LL
channels:  No channels selected yet
baselines: JbJb WbWb EfEf McMc NtNt OnOn ShSh TrTr YsYs SvSv ZcZc HhHh
sources:   No sources selected yet
time:      No timerange(s) selected yet
```

Each of the selection commands has a syntax of its own which typically allows for advanced data selection, specific to the type of interest. A lot of time has been invested in giving each selection command a means of selecting data based on 'logical' values rather than physical ones.

The next section should give some idea of what they can do.

For all discussed commands below the reader is cordially invited to read the online help inside *jplotter* for the command of interest. The **`help <command>`** tells all.

time

The **`time`** command allows selection of (multiple) time range(s) of interest. The symbolic constants **$start**, **$mid**, **$end** and **$t_int** (which dynamically evaluate to values for the current data set) can be used together with arithmetic and human readable time formats. This allows for natural-looking time selection commands like **`time $start to +1hr10m`** to select only the first hour-and-ten-minutes from the experiment. Also check the **`scan`** command further down for convenient scan-based selections. Multiple time ranges are separated by "**,**" (a comma):

```
jcli> time $start to + 5 * $t_int ,  $end - 1h to +20m
time:       06-Mar-2014/12:30:00.500 -> 06-Mar-2014/12:30:05.500
time:       06-Mar-2014/14:29:59.481 -> 06-Mar-2014/14:49:59.481
```

bl

The baseline selection command **`bl`** was already introduced earlier. It supports the pseudo baseline names **auto** and **cross** which select what one would imagine. "All baselines to Ef" becomes **`bl ef*`**. Or try this: "all baselines to either Jb or Hh": **`bl (jb|hh)*`**

The **`bl`** command supports multiple arguments. Each argument is a 'selector' and selects one or more baselines. The arguments are evaluated from left-to-right and a set of selected baselines is built. Each selector can add (default) or subtract baselines from the set so far. All cross-baselines to Ef would easily be selected as **`bl ef* -auto`**. The minus sign indicates to remove baselines matching the selector from the set.

src

The **`src`** source selection command is much like the **`bl`** command: it compiles a set of sources (de)selected by the argument(s) to the command, evaluated from left-to-right. Being a text based selection mechanism, common UNIX shell wildcards ("*" and "?") are supported: **`src j19*`** selects all sources whose name start with 'j19' (case insensitive). A limited form of regular expression syntax is supported: **`src 3c(18|192)`** selects both sources 3C18 and 3C192.

fq

This is the main subband/polarization selection command. Subbands are numbered *0..n* (as relabelled by *jplotter*[4]) within a frequency setup or group. For most data sets it will present the frequency information in a natural manner, to wit the output of the **`r`** command.

A bit was unveiled earlier when only the parallel polarization combinations for all subbands were needed. The very short **`fq */p`** effected just that. The **`fq`** command also accepts multiple arguments to select multiple, specific, subband+polarization combinations. For example **`fq 1,2/rr 5:7/l*`** would select the RR polarization from subbands 1 and 2 and LL and LR from subbands 5 through 7 (inclusive).

ch

The frequency selection is completed by the **`ch`** command, to select channels inside all selected subbands (see the **`fq`** command above). Channels are numbered *0..n*. The pseudo values **first**, **mid** and **last** are defined, which can be used in simple arithmetic expressions. The author's favourite is the magic

`ch 0.1*last:0.9*last` which selects the inner 80% of the channels of each band, or this one: **`ch mid-2:mid+2`** to select just a few channels around the center.

---

[4] Read the help for both the **`r`** and **`ms`** commands, specifically about how meta data is dealt with in *jplotter* and what you can do to alter the behaviour in case it is not appropriate. ALMA Measurement Sets do come to mind.

8. Scan-based data selection

The previous section dealt with data selection methods that operate directly on the meta data of the MS. Sometimes that is just not adequate. A typical observation consists of a number of scans; separate observations on some source at a particular frequency.

indexr
listr

The **indexr** command will attempt to regenerate the list of scans from the MSs data. If this has run succesfully:

```
jcli> indexr
Running indexr. This may take some time.
indexr: found  29  scans. (use 'listr' to inspect)
```

then **listr** can be used to inspect the scans **indexr** has found (the output has been snipped for brevity; not all 29 scans are shown):

```
jcli> listr
  1: 06-Mar-2014/12:30:00.500   5m28.98s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
  2: 06-Mar-2014/12:36:00.500   5m28.98s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
  3: 06-Mar-2014/12:42:00.500   5m28.98s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
  4: 06-Mar-2014/12:48:00.500   5m28.98s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
  5: 06-Mar-2014/12:54:00.500   5m28.98s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
  6: 06-Mar-2014/13:00:00.500   5m28.98s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
```

It shows the basic properties of each scan like start time, duration, which integration time was used and which source was observed.

Other than cosmetics, having a list of scans in memory enables the very powerful **scan** selection command. In its simplest form it can be used to just narrow down the data selection to specific scan(s), selecting them by scan number:

```
jcli> scan 10-12 21 29
 10: 06-Mar-2014/13:24:30.500   4m58.98s dT: 1.00s 0422+004        (1) (ARRAY_ID 0)
 11: 06-Mar-2014/13:30:00.500   5m28.98s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
 12: 06-Mar-2014/13:36:00.500   5m28.98s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
 21: 06-Mar-2014/14:31:30.500   9m58.96s dT: 1.00s 0528+134        (2) (ARRAY_ID 0)
 29: 06-Mar-2014/15:25:00.500   4m58.98s dT: 1.00s 0422+004        (1) (ARRAY_ID 0)
```

In the not-so-simple form, the **scan** command implements a SQL-like syntax to extract time ranges from (a selection of) scans; those matching criteria, if given.

This command will select 10 seconds worth of data out of *every* scan (again the output's been snipped). You'll see that the 'scan duration' is now exactly 10 seconds:

```
jcli> scan mid-5s to mid+5s
  1: 06-Mar-2014/12:32:39.989   0m10.00s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
  2: 06-Mar-2014/12:38:39.989   0m10.00s dT: 1.00s 0414+009        (0) (ARRAY_ID 0)
```

Assume some observing system is guaranteed to yield valid data only in the last minute of a scan. The following 'query' selects the last 20 seconds of data out of every scan that is both on the target source 0422+004 and is longer than 1 minute (to make sure it has valid data in this hypothetical system). Apparently only three scans match:

```
jcli> scan end-20s to end where length>60 and field ~ '042*'
 10: 06-Mar-2014/13:29:09.481   0m20.00s dT: 1.00s 0422+004        (1) (ARRAY_ID 0)
 20: 06-Mar-2014/14:29:09.481   0m20.00s dT: 1.00s 0422+004        (1) (ARRAY_ID 0)
 29: 06-Mar-2014/15:29:39.481   0m20.00s dT: 1.00s 0422+004        (1) (ARRAY_ID 0)
```

9. Time and/or frequency averaging; impact of *solint* and *weight threshold*

A very good diagnostic plot is phase-versus-time. For data in the frequency domain this means that the phase across the band needs to be averaged. In the time (lag) domain, the phase is the phase of the central lag.

With sources that have low signal-to-noise, it may be necessary to perform time averaging of a number of spectra before a signal becomes visible in the amplitude-versus-channel plot.

avc
avt

jplotter supports both time and/or frequency averaging of data before plotting. The settings are controlled via the **avc** (**av**erage **c**hannel) and **avt** (**av**erage **t**ime) commands. For both three averaging methods exist:

None	No averaging is performed. Each channel (*-versus-time) or each spectrum (*-versus-channel) is plotted individually
Vector	The complex data are averaged before computing the desired quantity, e.g. amplitude or phase. The code computes ***phase(avg(data))***
Scalar	The average amplitude or phase is computed: ***avg(phase(data))***

Note that only under certain circumstances both time- *and* frequency averaging can be supported. Usually, if either is the *x*-axis of your plot and jplotter is instructed to average that down to a single point, the result is not going to be very helpful or even visible.

solint

An important variable for the time averaging is the *solint* value. It can be set using **solint**. It is named after its illustrous ancestor from the (Classic) AIPS VLBI data reduction package. Its value is either **none** or a **duration**. Depending on *solint*'s value, time averaging, if requested, does different things:

None	All data for each individual time range will be accumulated and averaged. Multiple time ranges can be selected through the **time** command or, more interestingly, via the **scan** command (see previous).
duration	The data will be binned in bins of **duration** length and then averaged. A new time stamp is computed as the center of the interval. Using this it is possible to 'rebin' data to make sure time stamps are aligned. Alternatively a high time resolution data set can be 'smoothed' to lesser data points.

wt

Weight **t**hresholding can be applied in the plotting. If not **none**, only data points having their **weight** >= **threshold** value will be considered for plotting, or, if set, averaging.

jplotter automatically chooses between WEIGTH or WEIGHT_SPECTRUM columns for the per-spectral point weight, giving preference to the WEIGHT_SPECTRUM, should that optional column be available in the MS. The WEIGHT column holds a weight per polarization product and is virtually extended to give each channel the same weight. This is in accordance with application behaviour as required in the MS version 2.0 defining document, AIPS++ Memo 229, previously referred to.

10. Tinkering with the layout, ordering, scaling, multi subband, colours etc.

All plots on the screen have a number of properties, the **p**lot **p**roperties. Much as the **`sl`** command gives an overview of the current selection, the **`pp`** command yields an overview of the current plot properties, like type, layout and averaging settings:

```
jcli> pp
plotType:                wt [weight versus time]
layout[wt]:              4 plots organized as 1 x 4
mark[wt/weight]:         (none)
averageTime:             None
averageChannel:          None
solint:                  None
weightThreshold:         None
new plots on:            FQ BL
```

`pt`
`lp`

Mentioned earlier, **`pt`** is used to set the plot type; **`lp`** lists the plot types. Examples are **amptime**, **anpchan**, **ampuv** for amplitude-versus-time, amplitude-and-phase-versus-channel and amplitude-versus-uvdistance.

`nxy`

All plot types come with a default layout, roughly appropriate for the type of plot. The **`nxy`** command can be used to quickly change (or inquire) the layout for the current plot type: **`nxy 3 5`** organizes plots in a 3 column by 5 row grid.

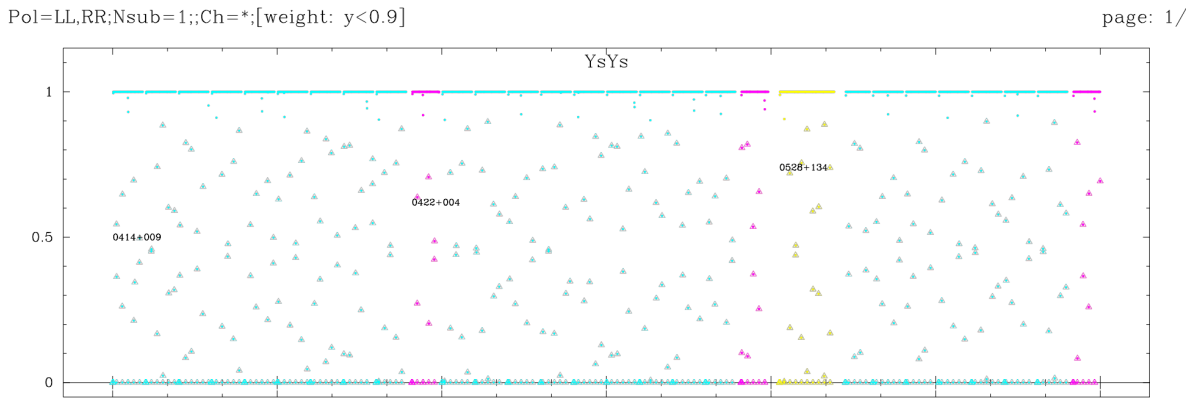
*-versus-time plots tend to be long, so they're organized in one column: **`nxy 1 *`**, allowing the full width of the display area for the time series.

*-versus-channel plots typically are organized in a two by four layout: **`nxy 2 4`**; spreading out a single spectrum across the width of the display area would just look … wrong.

`mark`

`mark`'ing data points can be used to mark data points in a visually distinct manner if they match a criterion of your choice. The mark command accepts a single-line boolean Python expression involving the variables *x*, *y* or both. For each data point *(x, y)* to be plotted, the expression is evaluated. All points for which the expression evaluates to **true** are plotted with a large triangle symbol, to make them stand out from the ordinary points.

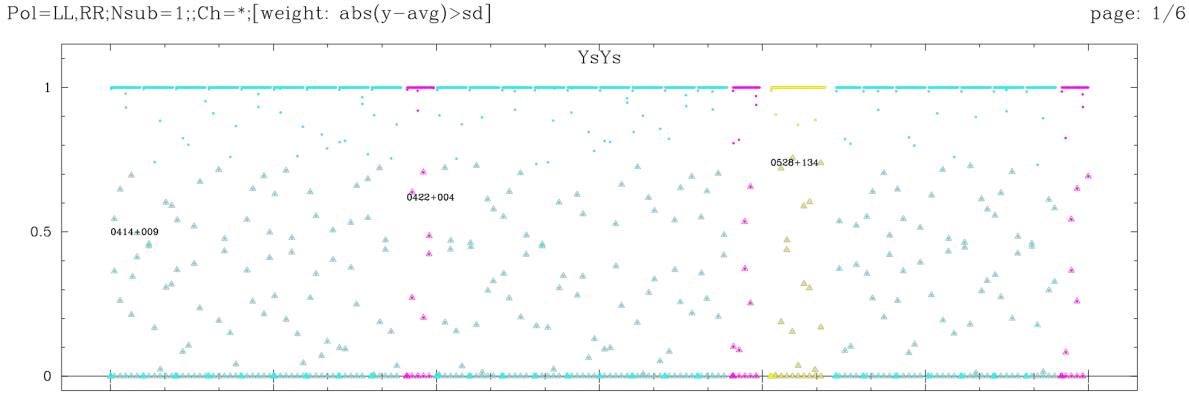
To see where the weight is less that 0.9, a simple **`mark y<0.9`** would do:



But the marking goes deeper. For each separate data set, if marking is requested, the system computes the average and the standard deviation. The results of this are made available in the expression evaluation environment through the pre-defined variables **avg** and **sd**. Their value will always evaluate to the respective value for the data set the *(x, y)* point under consideration is from.

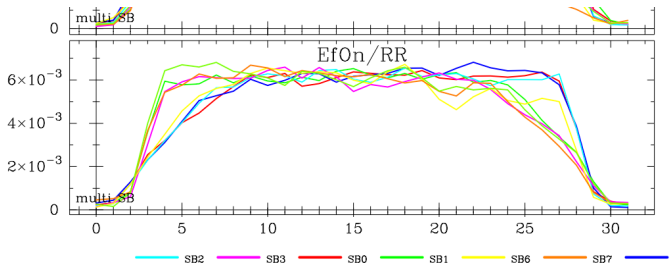
Furthermore, all functions from the Python **math** module are made available without module name prefix, like **abs()** and the trigonometric functions **sin()**, **cos()** etc.

To illustrate: in order to mark those points that are more than one standard deviation off with respect to the average, the following expression could be used: **`mark abs(y-avg)>sd`**:



`new`

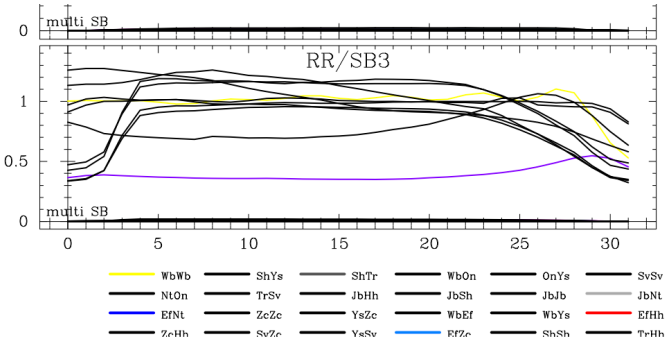
The role of the **`new`** command has been briefly explained in section **5.** earlier. Having the possibility to flexibly set plot ⇔ dataset ordering criteria gives the user full control of how the data sets should be (over)plotted. Sometimes it is desirable to compare the individual polarizations for the selected subbands whilst another time it might be more illustrative to compare the individual subbands' polarizations, as illustrated below. In both plots below spectra (**`pt ampchan`**) are overplotted, only according to different **`new`** plot settings.



This compares the spectra for all subbands on each baseline + polarization combination.

The **`new`** command to provoke this is:

```
jcli> new all false bl,p true
new plots on:                BL P
```



This plot compares the baseline responses for each subband + polarization combination.

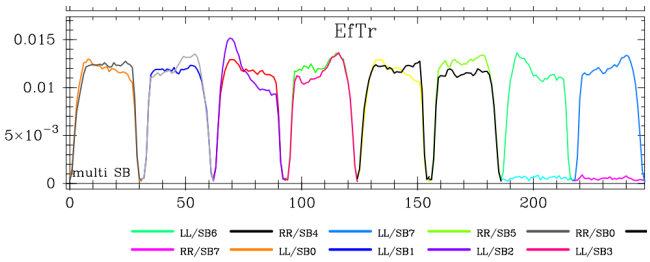
The new setting for this was:

```
jcli> new all false sb,p true
new plots on:                SB P
```

multi If multiple subbands are selected, then by default they will be drawn *on top of* each other in the same plot[5]. But more often then not it's more informative if the whole (selected) spectrum is visualized i.e. that the individual subbands are visible *next to* each other. Which presentation is best primarily depends on what the intent of the comparison or visualization is.

There are two ways out of this conundrum:

1. add the subband label to the 'new plot' settings, such that each subband is plotted in a separate plot: **new sb true**, however that may not be as easy on the eyes. (But see the **sort** command later on.)
2. set **multi** to **true**. When **multi** is **true** then multiple subbands drawn in the same plot will be plotted next to each other in stead of on top of each other. **multi**'s default value is **false**. Again the same data set was used as in the previous example (explaining the **new** command).



This plot was made with the following settings:

```
jcli> new all false bl true
new plots on:  BL
jcli> multi true
multisubband[ampchan]: True
```

sort Usually multiple plots are created from a selection. Each plot is labelled with the unique values of the labels as set in the 'new plot' command **new**. *jplotter* draws the plots in random order, filling pages as necessary. This makes for difficult comparison; specific plots cannot easily be looked up. Use of the **sort** command can overcome this by allowing the user to explicitly set a plot sorting order. The arguments to the sort command are, once more, one or more label types (see section **6.**) - plots will be sorted by those labels. If more than one label types are given, the plots are first sorted on the first label, secondly on the second etc.

Examples:

The following sort order groups all subbands together per baseline such that all subbands on the same baseline can easily compared:

```
jcli> sort bl sb
```

whereas the this groups all baselines together by subband, such that for each subband the different baseline responses can be compared:

```
jcli> sort sb bl
```

draw Some plot types look better drawn with **lines** (e.g. a spectrum) whilst phase versus time looks better when drawn with **points**. Using the **draw** command it is possible to tell *jplotter* to **draw** the plot type with either **points**, **lines** or **both**. This setting is kept per plot type. See also the **ptsz**/**linew** commands described below.

[5] if you haven't played with the **new** command yet; see previous

ptsz
linew

If the default point-size when drawing the plots with points or the width of the lines when drawing with lines are not to taste, add a bit extra using the **ptsz** or **linew** commands. The argument to both commands is just a number; the requested point size/line width (this should, however, not come as a total surprise).

y[01]
x

jplotter's default behaviour is to give each plot the same scale such that the plots can be directly visually compared. Under certain circumstances this may be undesirable, e.g. when both auto- and cross-correlation spectra are plotted.

Using the **x** and **y** command the scaling of the indicated axis of the plots can be set to either of the three values as described in the table below. In multi-panel plots (e.g. amplitude and phase versus *), the scale of the individual panels can be set using the **y0** command for the bottom panel or **y1** for the top panel.

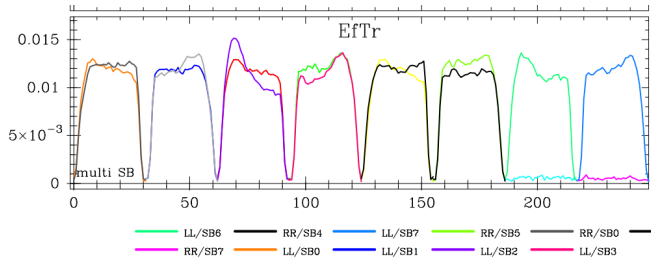
The scaling settings are kept on a per-plot type basis.

global	The scale of each panel is set to fit the global minimum and maximum over all data sets displayed in all plots.
local	Each panel will be scaled to fit the minimum and maximum of all data sets displayed in that panel.
<min> <max>	Set an explicit scale by manually providing a minimum and maximum value. All panels get this scaling.

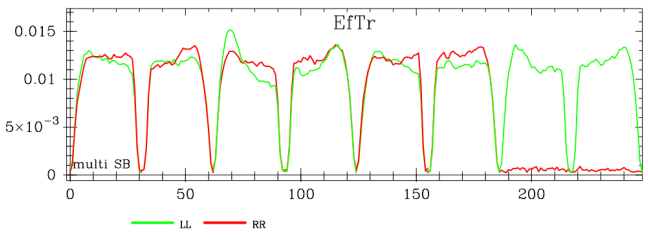
ckey

Another command that combines a mini-language and label types. *jplotter*'s default data set colouring algorithm is quite simple: each unique data set label gets its own color. Again, not always is this the best choice. Specifically for those occasions *jplotter* allows (very) fine grained control over the data set colouring algorithm via the **ckey** (**c**olour **key**) command. If desired specific values can be given a specific colour. The syntax is documented in the **ckey** help documentation.

The two plots below should give an idea of what the **ckey** command does. The same plot as under the description of the **multi** command is used.



Each data set - a combination of a subband and a polarization - has its own colour. Presented like this, the colours per subband do not add information whilst at the same time it is difficult to see which polarization(s) fail in subbands 6 and 7.



After issueing the following command:

```
jcli> ckey p
```

the plot looks like this. This instructs *jplotter* to colour the data sets by **p**(olarization) only and the affected polarization is readily identified.

11. Multi window/batch support, save plots to file, navigating pages of plots

The default graphics device is output to an X11 window. In the PGPLOT library, X11 windows can only be identified by *number*. The default graphics device is the X11 window with number **42**.

jplotter supports having multiple windows open at the same time. Due to restrictions in the PGPLOT library this does not, however, extend to files: there can be only one file open for plotting into at any instance of time.

Internally *jplotter* keeps plotting environments. Each individual environment has its own graphics device, MS, selection, 'new plots' and plot type. Changes in either have no effect on the settings in other 'windows' (let's call them that for the moment). Note that changes in the plot properties *do* affect the same plot type in other windows, if they are replotted.

win

The **win** command either displays the current window that processes the commands or switches control to the indicated window. If the indicated window did not exist yet, a new one will be created with no Measurement Set opened and an empty selection. In order to display data into a freshly created window not always a Measurement Set has to be opened; a previously **store**d set of plots can be **load**ed with a different 'new plots' setting to name but one way.

i, f, l
[<j>]p
[<j>]n
<j>

It is not uncommon for *jplotter* to produce multiple pages of plots. Navigation can be done by jumping to the j^{th} **n**ext page *j***n** (or *j***p** for j^{th} previous). The number *j* is optional and defaults to 1 (one). **f** goes to the first page, **l** (lower case ell) to the last. Just typing a number, *j*, jumps to that page. The **i** command is for **i**nteractive mode. The **f**, **l**, **p**, **n** commands can be used as if typed at the command prompt (if the plot window is the X11 window with the focus). On systems where the mouse is supported a right-click in the window moves to the next page, a left-click to the previous page. Typing **q** in the window exits **i**nteractive mode.

file

Switches to or creates a plot file for output. The argument to **file** is the file name of the output file name. *jplotter* could handle multiple file(s) to be open at the same time if the PGPLOT library would. By switching to an output file before beginning plotting, output will go to file directly, without having to go through a graphical window. This can be useful for batch plotting.

refile

With the **refile** command it is possible to redirect the output of the current environment to a new file, without creating a new environment. As such the Measurement Set, selection, 'new plots' etc. remain unaffected. Any previous graphics device associated with the environment is closed.

close

Can be used to **close** the current plot device. This can be an alternative to the **refile** command - it allows plotting to a different file. Contrary to the refile command, the **close** command takes down the whole environment with it.

save

The **save** command allows storing the current plots in a PostScript file. By suffixing the file name with **/vcps** portrait mode is available; landscape (**/cps**) is the default.

12. Miscellaneous: macro's, play a script file, write out selection to new MS

jplotter offers two methods to simplify repetetive commands: **macro**s and script files.

macro

macros are nothing but text substitutions and their use is limited, although *jplotter*'s "smart" selection commands allow useful, MS detail agnostic, **macro**s.

E.g. the following macro, when called as **mk_phatime** (after it's been defined), plots the phase-versus-time of the vector averaged inner 80% of the channels on all subbands on the cross baselines and makes sure the plots are drawn with points.

```
jcli> macro mk_phatime 'pt phatime; bl cross; fq */p; ch 0.1*last:0.9*last;
avc vector; avt none; draw points; pl'
# … time passes …
jcli> mk_phatime
# … plots are made and drawn on the screen: profit!
```

The nice thing is that this macro works unmodified on almost all (VLBI) Measurement Sets.

Note that the macro definition is enclosed in single quotes to avoid the command interpreter executing the command(s) rather than putting them into the macro's definition. Macro expansion is recursive; macros can expand to text containing other macros etc.

play

Another way to store a sequence of plot commands is to put them in a text file which can be **play**ed. In direct contrast to macros, the script can be passed arguments when it is **play**ed. The way this works is that each (uncommented) line of input in the script file becomes a Python string-formatting string. Any occurrences of the form **{<number>}** will be replaced with argument number **<number>** that was passed to the script, starting from 0 (zero) for the first argument.

The previous macro could have been put in a file, e.g.:

```
verkout@eee:~$ cat mk_phatime.jpl
# script to create phase-versus-time plots to reference antenna
# needs two arguments: the MS name {0} and the refant {1}
ms {0}
# set up plot - make sure points are used to draw
pt phatime; draw points
# select cross baselines to the refant, parallel hand
# polarizations and 80% of the channels
bl {1}* -auto; fq */p; ch 0.1*last:0.9*last
# make sure averageing settings are correct
avt none; avc vector
# and execute
pl
```

Besides being more readable and flexible, the script can be commented, which is always a nice bonus.

The script can then be **play**ed from within *jplotter* as follows:

```
jcli> play mk_phatime.jpl EA053D.ms ef
```

write

The current selection (**sl** to view current selection), ignoring channel/polarization selections, can be written out to a new Measurement Set using the **write** command. This will create a MS with referenced data: the new MS will refer to rows in the MS whence the data came. Because it's row based, the channel/polarization selections cannot be honoured; these would select inside a row's data matrix.

13. Arithmetic expressions with plots - straight difference or dividing or …

jplotter can **store** (the evaluated value of) *expressions*, potentially involving previously stored variables, as named variables:

```
jcli> store 3.14 as pi
jcli> store 2*pi/360.0 as deg2rad
```

Variables are named with an *identifier*; the first character must be a letter; subsequent allowed characters are the alpha-numericals (i.e. letters and digits).

Besides simple values, *jplotter* can **store** the (value of an *expression* involving the) current set of plots. What happens is that *jplotter* will evaluate the *expression* for each individual data set. If an *expression* involves two sets of plots or more[6] - e.g. the simple difference "*a* - *b*" - only the data sets within those sets-of-plots with identical label values are combined.

Just as when there is smoke it is implied that fire can usually be found in its vicinity, likewise where there is **store** then **load** is generally not too far off.

Without an argument, **store** (as well as **load**) print the currently defined variables:

```
jcli> store
Currently defined variables:
pi         = 3.14
deg2rad    = 0.0174444444444
_          = None
```

The current-set-of-plots does not have a name but is identified by the special variable '_', the underscore. Apparently, in this *jplotter* session, no plots were yet created.

The '_' is the "default argument" in the **store** and **load** commands. It is either the

store *expression* **as** *name*
load *expression*

expression **or** the *name* in the **store** command, but not both:

store *expression*	is short for	**store** *expression* **as _**
store as *name*	is short for	**store _ as** *name*

The **load** command explicitly loads the result of *expression* into the '_' variable.

After loading a value into the '_' variable that describes a set of plots, the resulting plots are drawn in the current graphical device using the current setting for 'new plots'. If the result was not a set of plots, nothing happens.

In these examples it is assumed that a MS was already opened and some plots created:

```
jcli> store as old_data   # store current set of plots as old_data

# Open other MS, redo selection, make plots of that …
jcli> store as new_data   # store the new plots as new_data

jcli> store               # inspect what variables we have
new_data   = 'ampchan' from EA053D-no0021-sfxc.ms [11 plots]
old_data   = 'ampchan' from EA053D-no0021-sfxc-prod.ms [11 plots]
_          = 'ampchan' from EA053D-no0021-sfxc.ms [11 plots]

# Let's open a new window
jcli> win 44
jcli> store old_data - new_data as diff  # store result of expression
jcli> load diff                          # plot straight difference
jcli> load old_data / new_data           # or compute the ratio old/new
```

[6] *jplotter* knows if a variable refers to a set of plots or not

13. Putting it all together: fancy spectra plots (e.g. plot on page 1)

Let's finish off with an annotated recipe for reasonably good looking and informative diagnostic plots, using quite a few of the commands discussed above.

This sequence of commands computes time-averaged spectra of the parallel hands of polarization of all subbands (however many there are) by scalar averaging 10 seconds of data from each scan, giving the RR polarization the red colour and the LL polarization the green colour.

Because we plot both auto-correlation as well as cross-correlation spectra the *y*-axes of the plots must be individually scaled; the cross-correlation amplitudes typically are significantly lower than the auto-correlation amplitudes.

The plots are sorted by baseline and then by time such that for each baseline the scan's performance through the different scans can be easily checked.

```
# open the MS and index it
jcli> ms EA053D.ms; indexr

# we want time averaged spectra. These look best with lines
# and with all subbands next to each other. Set y-scaling
# to local on account of both auto- and cross spectra
jcli> pt ampchan; draw lines; multi true; y local

# set time averaging, set solint to none such that each scan gets
# averaged separately
jcli> avc none; avt scalar; solint none

# do the selection - just select the parallel hand polarizations
# and 10 seconds out of the middle of each scan
jcli> fq */p
jcli> scan mid-5s to mid+5s

# we want a new plot for each baseline and scan (i.e. TIME)
# and the plots sorted by baseline and time too
jcli> new all false bl,time true
jcli> sort bl time

# we want to color the data sets by polarization;
# specifically RR=red, LL=green
jcli> ckey p[rr]=2 p[ll]=3

# Let's go!
jcli> pl
```

Appendix A.

jplotter depends on two external Python packages:

pyrap - for accessing the data stored in MS format. *pyrap* is the Python binding that comes with casacore < 2.0. At this time there is no support for the *python-casacore* binding that comes with casacore versions >= 2.0.

pyrap	`https://code.google.com/p/pyrap/`
casacore < 2.0	`https://code.google.com/p/casacore/`

ppgplot v1.4 - a Python binding to the PGPLOT graphics subroutine library. Although ppgplot versions can be found all over the interwebs (e.g. `https://github.com/npat-efault/ppgplot`) these are typically version 1.3 of the Python binding.

The author hosts a copy of the ppgplot-1.4 source code at:

`http://www.jive.eu/~verkout/ppgplot-1.4.tar.gz`

Future revisions of this document will likely contain more details about installing the dependencies and the dependencies of those …