Design of casa::TablePlot for Casapy

Urvashi Rau

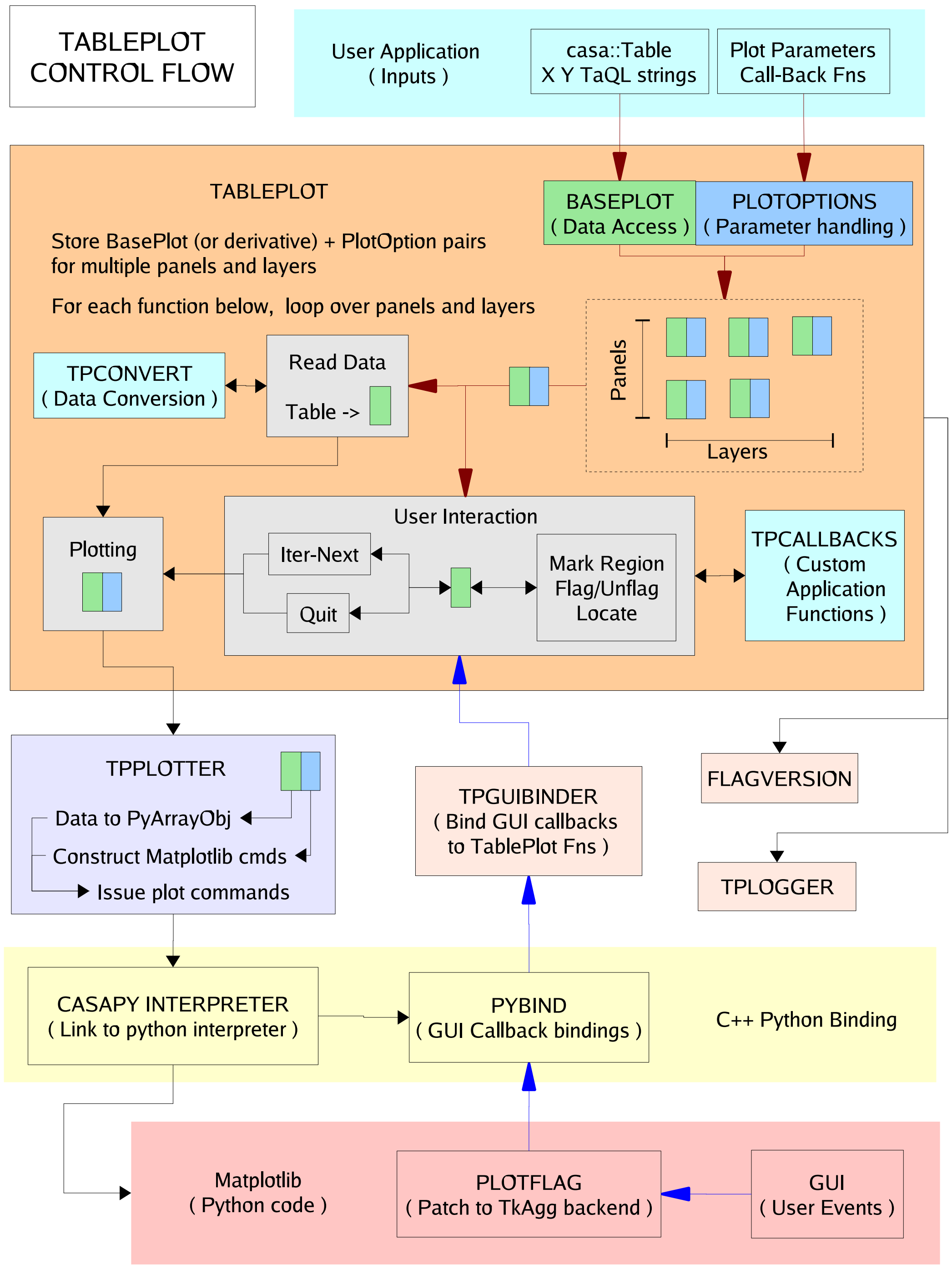
18 Dec 2007

Abstract

This note describes the design of the `casa::TablePlot` classes and the control flow through them for data access and plot creation. The use of the Matplotlib plotting package is described, along with a performance analysis to quantify bottlenecks.

Many thanks to S.D.Jaeger, G.Moellenbroek, and K.Golap for some bug-finding/fixing, and helpful suggestions about design changes, during their development of the application level classes that use `casa::TablePlot`. The work done on this code was funded via a Graduate Student Research Assistantship.

Contents



TABLEPLOT CONTROL FLOW
User Application (Inputs)
casa::Table X Y TaQL strings
Plot Parameters Call-Back Fns
TABLEPLOT
Store BasePlot (or derivative) + PlotOption pairs for multiple panels and layers
For each function below, loop over panels and layers
BASEPLOT (Data Access)
PLOTOPTIONS (Parameter handling)
Panels
Layers
TPCONVERT (Data Conversion)
Read Data
Table ->
User Interaction
Iter-Next
Quit
Mark Region Flag/Unflag Locate
Plotting
TPCALLBACKS (Custom Application Functions)
TPPLOTTER
Data to PyArrayObj
Construct Matplotlib cmds
Issue plot commands
TPGUIBINDER (Bind GUI callbacks to TablePlot Fns)
FLAGVERSION
TPLOGGER
CASAPY INTERPRETER (Link to python interpreter)
PYBIND (GUI Callback bindings)
C++ Python Binding
Matplotlib (Python code)
PLOTFLAG (Patch to TkAgg backend)
GUI (User Events)

Figure 1: Flow of control through a set of classes that provide 2D plotting functionality for data stored as casa::Table. Data is read from casa::Table and derived quantities are computed and plotted (black arrows). User interaction with the plotted data via queries and editing is accessible from the command line (maroon arrows) as well as from the plotter GUI (blue arrows). Custom computations are allowed via optional callback functions defined by the user application (light blue boxes). The pink and yellow layers at the bottom control the C++ Python binding and plotting package specific code.

1 Classes and control flow

`casa::TablePlot` is a Singleton class that manages plot requests from multiple applications (`casa::MSPlot`,`casa::PlotCal`,`casac::tableplot`). It receives input in the form of data Tables and Plot-Options for each plot and manages multiple panels on the plot window with multiple plot layers per panel. It maintains and sends commands to a single instance of a plotter GUI, and responds to user interaction both from the GUI and the command line.

1.1 Data Access

1. **Table Access :** Columns from any kind of casa::Table can be plotted against each other. Input tables can be Measurement Sets, MS Subtables, Calibration tables, etc.., or reference sub-sets of Tables generated via any Table selection mechanism. Multiple tables of different types can be opened simultaneously for plotting and interactive editing. Data sets can be iterated through to generate a series of plots from different subselections. Columns with finite and ordered values can be iterated upon. Iteration rules follow that of the casa::TableIterator class. Iterations can proceed only in one direction - forward.

2. **Expression Evaluation :** Expressions for derived quantities are written using the TaQL[1] syntax. For data from ArrayColumns, in-row selections within the Array are done via TaQL indices (for example : `AMPLITUDE(DATA[1:2,1:10:1])`). For sample TaQL strings, please refer to Appendix A. Quantities that cannot be computed purely via TaQL expressions make use of conversion functions supplied by the application in the form of callbacks `casa::TPConvertBase`.

3. **Data extraction :** `casa::BasePlot` is responsible for evaluating TaQL expressions for X and Y axis data, performing conversions on the results, and storing the data in arrays to be plotted. It provides a set of query functions that the plotter class `casa::TPPlotter` will use to access data to be plotted. Derivatives of `casa::BasePlot` can implement customized data access, but must provide the same view of the final data to be plotted, in the form `casa::Array`s and query functions that `casa::TPPlotter` expects. `casa::CrossPlot` is one derivative, used for plots where the X data comes from ArrayColumn Array indices (for example, channel number), and not TaQL string evaluations. Other derivatives can be implemented to read data without using TaQL expressions. Note that the rest of the framework does not depend on how the data is accessed internal to `casa:BasePlot`.

4. **Flag Handling :** Flags are read using the same in-row selection as the data itself, and stored in arrays. `casa::TPPlotter` uses the `casa::BasePlot` query functions to check flags with data and select points for plotting according to on-the-fly selection criteria (only unflagged, only flagged, every n^{th} point, average n points, points within a certain range of values). Interactive editing modifies the flags in the `casa::BasePlot` flag storage arrays. The `casa::BasePlot` (or

[1]Aips++ Note 199 : *http : //aips2.nrao.edu/stable/docs/notes/199/199.html*

derivatives) are then responsible for writing these flags back to the Table, along with any translations or flag-expansions (for plots of averaged values). It is always ensured that the `FLAG_ROW` column holds the logical `AND` of all flags per row in the `FLAG` column. The flags are written to disk after every interaction, to allow all currently visible plots using the same Table, to immediately reflect flag changes. Columns to be used for flags can be user specified, and default to `FLAG` and `FLAG_ROW`.

1.2 Plotting

1. **Plot Parameters :** `casa::PlotOptions` manages user input. It contains defaults for all parameters, and functions to validate parameters and do error checking. Parameters are divided into those common to all layers on a panel (panel location, size, axis labels..), and those that can vary across layers (plot colour, marker format...), and stored in a wrapper `casa::PanelParams` class. For a list of plot parameters and their functions, please refer to Appendix B.
2. **Panel management : For each panel,** `casa::TablePlot` maintains an instance of `casa::PanelParams` and a list of `casa::BasePlot`s. Plots from multiple Tables, and overplots can generate multiple layers per panel. This class iterates through panels and layers and sends `Vector<casa::BasePlot>, casa::PanelParams` pairs to a single instance of `casa::TPPlotter` for plotting.
3. **Plotting :** `casa::TPPlotter` performs two tasks. First, it receives a list of `casa::BasePlot`s, and queries each of them for the number of plot commands to run, and the number of points per plot. For each plot command, it reads data and flags through `casa::BasePlot` query functions, and assembles the selected points into plotting-package-specific data format. Then, it reads plot options from supplied `casa::PanelParams`, and constructs plot commands. Finally, the data and plot commands are combined and sent to the plotting package.

1.3 User Interaction

The following functions allow the user to interact with any displayed plot. They are accessible from the command line by directly calling `casa::TablePlot` functions as well as from buttons on the GUI which internally call the same `casa::TablePlot` functions. These functions operate on the stored `Vector<casa::BasePlot>, casa::PanelParams` pairs per panel, and trigger refreshed plots, as well as the transfer of flags to and from the Table on disk. A no-GUI mode of plotting has also been provided. In this mode, the plots are created as described above, but are not rendered onto a plot window. Instead, additional commands can be used to directly save a plot into a file on disk.

1. **Mark Regions**:
   Rubber-band boxes can be drawn to mark rectangular regions on a plot. The same effect can be reached by sending in world-coordinate box specifications from the command line.

2. **Zoom/Pan**:
   The matplotlib TkAgg GUI provides buttons for Zoom/Pan modes. (Note that Matplotlib stores a full copy of all data points currently on the plot, in Double precision, to enable Zoom/Pan modes directly from the GUI.) Command-line interaction can be done via the `plotrange` plot option. This interface can be used if a new plotting package does not hold all data points in memory for Zoom/Pan.

3. **Flag/Unflag**:
   Buttons for flagging and unflagging have been added to the TkAgg matplotlib GUI. Their callbacks trigger `casa::TablePlot` functions that modify values stored in `casa::BasePlot` flag arrays, which then get transferred to the Table on disk. If multiple views of the same data are plotted on different panels, flags from one panel are reflected in the others. `casa::TablePlot` returns a flag summary as a `casa::Record` which holds all information necessary for recreating each flag/unflag command, and can be used to generate a flag history. Currently this information is sent to the logger, but it can be returned as python records for scripting.

4. **Locate**:
   A GUI button has been added for the operation of obtaining meta-data about selected points. Table row indices are inferred from the selected data points, and values of specified Table Columns (or TaQL expressions) for only these rows are read and displayed. One entry is produced per selected row. The control flow is the same as for Flag/Unflag, and a record of meta-data is returned by `casa::TablePlot` for the selected points.

5. **Iter-Next**:
   A GUI button was provided to move to the next plot in an iteration series.

6. **Quit**:
   The GUI has been provided with a button that triggers the destruction of the plot window, as well as the clean-up of C++ data structures associated with all currently visible plots. The X button on the GUI window has also been bound to the same function to ensure the proper release of C++ allocated memory, when the plotter GUI is shut.

For custom operations required by the applications, callback functions can be provided via the `casa::TPGuiCallBackHooks` and `casa::TPResetCallBack` classes. The `casa::TPLogger` class handles all logging operations for the plotter classes. The plot window can be accessed from the `Casapy` command line, for users to set custom labels, markers and legends, and to overlay arbitrary plots on top of existing ones.

1.4 Flag Versions

`casa::FlagVersion` is a stand-alone class that manages flag versions by creating named disk copies of `FLAG, FLAG_ROW` columns from the main table. Flag versions can be saved and restored along with options of merging multiple flag versions via boolean operators. Flag version tables are currently created in a directory parallel to the parent Table.

2 GUI and Python

This section describes the C++ / Python interface used for plotting.

2.1 Design Constraints

The main constraints that drove this design were as follows.

1. Using Matplotlib : After implementing interfaces for PGPLOT and PLPLOT and finding functionality restrictions, Matplotlib was chosen. It would have the added advantage of being accessible and controllable from the casapy command-line.

2. Multiple Interfaces : We needed to attach the C++ classes directly to the `casapy` python interpreter so that the GUI (and its event loop) could be controlled from the C++ code as well as from the command line. The XML interface provides control only from Python to C++ but not the other way around.

3. Memory Mapping : Direct memory mapping of C++ PyArrayObjects to python variables was required, but not possible at the time via the XML C++ to Python conversion interface. Therefore, instead of implementing a pure Python function for all GUI handling, methods descibed for standard Python Extending and Embedding [2] were used to implement direct C++ to Python binding with memory-mapping of data arrays as variables in an internal view of the `Casapy` interpreter. This binding however could not be done from within the `casa` namespace, so the binding code was made global.This required an additional step to send signals to classes within the `casa` namespace.

4. Enhanced GUI : The standard Matplotlib backend does not provide all the functions we require, and needed to be augmented (add buttons for flagging and locating, and implement callbacks for rubber-band region marking). However, for maintenance reasons, this had to be done without touching the standard Matplotlib distribution code tree. A run-time patch to the GUI backend was therefore developed.

2.2 C++ to Python binding (plotting)

1. `casa::CasapyInterpreter` uses the Python/C API to send python command strings to an instance of a python interpreter. In this case, the interpreter is the same as the casapy interpreter that the user sees as the command-line, but with only `pylab` visible. `casa::TPPlotter` sends command strings to this class, to be sent on to python.

2. `PyBind` implements the C++ to Python memory mapping required for data arrays stored as PyArrayObject in `casa::TPPlotter`, to be visible as python variables that can be used in Matplotlib plot commands.

[2]`http://docs.python.org/ext/ext.html`

The control flow for plotting is as follows : `casa::TPPlotter` sends plot commands ⇒ `casa::CasapyInterpreter` sends the commands to python ⇒ `PyBind` memory maps the data arrays to python variables ⇒ Matplotlib plot commands are executed.

2.3 Python to C++ Callback binding (user interaction)

1. `PlotFlag` is a runtime python patch to the Matplotlib TkAgg backend. It adds buttons to the GUI, and implements event handlers for these buttons that call `PyBind` functions (`PyBind` is visible from the python interpreter). It also augments the handlers for existing GUI buttons (Zoom/Pan/Forward/Back/Home), so that they and the new buttons would work seamlessly together in the Tk event loop.
2. `PyBind` also implements Python to C++ callback functions that are triggered by the GUI buttons via `PlotFlag`, and which transfer information into C++. However, `PyBind` is global and needs a way to access `casa::TablePlot` user interaction functions.
3. `casa::TPGuiBinder` provides the link between `PyBind` and the `casa` namespace by being global, but being instantiated from inside `casa::TablePlot`.

The control flow for event handling is as follows : GUI Button Press ⇒ `PlotFlag` Event Handlers ⇒ `PyBind` python-to-C++ functions ⇒ `TPGuiBinder` for global to namespace::casa transfer ⇒ `casa::TablePlot` user interaction functions.

3 User Applications

Two applications currently use the `casa::TablePlot` classes.

1. `casa::MsPlot, casac::msplot` : Applies selections to Measurement Set Tables to create subset Tables to pass on to `casa::TablePlot` for plotting, constructs TaQL expression strings for a large number of commonly generated plots, implements conversion functions for derived quantities that cannot be expressed as pure TaQL strings and custom callbacks that are called at the end of each user-interaction operation.
2. `casa::PlotCal, casac::calplot` : Applies selections to Calibration Tables, constructs TaQL strings for standard plots, and manages conversion functions and cleanup callbacks for user-interaction.

The `casac::tableplot` tool provides a lightweight interface to the `casa::TablePlot` class. All these applications communicate with a single instance of `casa::TablePlot`.

4 Performance analysis

4.1 Timing and Memory Usage

Timings and memory usage measurements are reported and discussed for the following dataset. (Tests were run on ”ballista” in April 2007.)

700 MB dataset :
DATA column = 2 correlations x 7 channels x 1876352 rows = 26 million points.
= 200 MB (stored as Double or Complex)
= 115 MB (100MB for Y-axis data (Float) + 15 MB for X-axis data (Double))
Disk reads/writes of boolean flags (3MB) during user-interaction do not exceed 5 sec.

1. **Disk read time**: Measured 30 sec. Expected 10+ seconds ((200 MB on disk)/(20 MB/s raw disk read speed)). The DATA column is read row by row (random access) followed by arithmetic operations on each Complex value `"SQRT ( REAL**2 +IMAG**2 )" or "ABS"`. The time spent on reading by TaQL expression evaluation was tested and found to be comparable to extracting Complex values row by row and separately performing the evaluation. The bottleneck was in reading row-by-row (20 sec for raw reads) versus reading the entire TableColumn at once (sequential access : 7 sec for raw reads).
   Problem : TaQL cannot be used in getColumn mode. It needs to be done row by row. Faster data access mechanisms can be used as derivatives of `casa::BasePlot`.

2. **Plotting Time**: Measured 60 sec. Expected 52 sec ((26M points)*(500K points/s for native matplotlib)). The break-up of the plotting time is as follows.
   Transfer data from `casa::Array` to PyArrayObject : 8 sec
   Matplotlib command `pl.plot()` : 12 sec
   Matplotlib command `pl.draw()` : 40 sec
   An attempt was made to eliminate the 8 seconds, by using `MaskedArrays` in python (to handle flags), and doing memory-mapping between `casa::Array` and `PyArrayObject`. The python plotting memory consumption went even higher since it duplicated the flags for X and Y as Integers.

3. **C++ Memory Usage**: Measured 150 MB. Expected 115MB for data + few MB for bookkeeping.
   Total memory usage at startup of `Casapy` : 120 MB
   Total memory usage while reading from disk : 120+150=270MB

4. **Matplotlib Memory Usage**: Measured 400MB. Expected 400MB (26M points) *(double precision for X and Y). Total memory usage while plotting : 120+150+400 = 670MB. The bottleneck is that Matplotlib stores an extra copy of all X and Y values in Double Precision (even if memory mapping is used between `casa::Array` and `PyArrayObject`). This amounts to 400MB = 2 $\times$ 200MB for 26 million X and Y values in Double precision.

The major bottlenecks are the Matplotlib rendering time and the Matplotlib backend memory usage. The choice of a new plotting package must fix this !

4.2 Optimization strategies

1. **Re-use of data :** When a plot already exists, data is re-read from disk only if there is a change in the data being plotted. Benign plot option changes (for example, plot colour) between plot commands do not require re-reading data from disk. Flags however, are re-read everytime a plot is made, but reading/writing the entire column of Booleans is not expensive, and it provides the feature that all plots from the same Table will immediately reflect flag edits done on any of the plots. (Note : The current task interface in casapy over-rides this optimization strategy by forcing a Table read for every plot. It does so,in order to conform to the task format of opening and closing tools in every run.)

2. **Numerical precision :** Float is used for all data stored for plotting. Expression evaluation and conversions are done per value in Double precision before converting to Float. The X-axis data is always stored in Double precision, to allow for handling MJD time values that require more than 7 significant digits in precision. When a plot involves a single set of X values, for multiple Y values, (for example, amplitude of 10 channels of data vs uvdistance), this is detected and only one instance of X values are stored in memory.

3. **Array to PyArrayObject :** Data is stored in large Arrays in `casa::BasePlot`, but are sent to Matplotlib in pieces, to minimize the amount of data being held as a PyArrayObject copy. Direct memory mapping between `casa::Array` and `PyArrayObject` was not done, because of the non-contiguous nature of points to be plotted (flagged, unflagged, every n^{th} point, etc). Tests were done with using Python Masked Arrays (to handle flags) along with memory mapping, but Matplotlib held duplicates of the flags as well and increased the memory usage beyond the saving obtained via memory mapping.

4. **Flagging:** Flag commands update `casa::Array`s of flags in memory, which are written to disk only once after all flag regions have been serviced.

5. **Locating :** Meta data for selected points is obtained by accessing the Table on disk only for rows corresponding to selected data points. The in-memory arrays provide enough information to isolate this subset of rows.

6. **Averaging :** In case of data averaged within a row (channel averages), flag changes must be reflected in the averaged data being plotted, and this re-evaluation is done only for selected rows for which flags have changed, instead of re-reading everything from disk. Since flag commands usually flag a very small fraction of the data, this saves time. In case of data averaged across rows (time averages), the averaging is done on-the-fly as part of data point selection while filling in PyArrayObjects. This allows multiple average-intervals to be specified as a plot option, and be replotted without having to re-read from disk.

5 Miscellaneous

5.1 Known problems

During the development of this set of classes, the requirements for the final product have often changed. For the most part, the original design was able to handle the new requests, but there are some known problems with the current implementation.

1. The extra memory copy in the matplotlib backend (in Double precision) is a bottleneck for interactivity. This copy is stored mainly to allow Zoom/Pan functions within the GUI. A different GUI that does not provide Zoom/Pan would solve this problem, and Zoom/Pan can be implemented via the `plotrange` plot option.

2. `casa::BasePlot` tries to read all selected Table rows at once. A more efficient way would be to iterate through the data (perhaps using `casa::VisibilityIterator` for MeasurementSets). Alternatively, to store all flags in memory for efficient interactivity while flagging, the use of `casa::TempArray` might be a better way to reduce the memory footprint.

3. Scalar averaging across rows requires the pre-division of the main Table into subtables on which it is appropriate to average across adjacent rows. For scalar time averaging on a Measurement set, this requires the subdivision into a subtable for each field, spectral window, scan, and baseline. This subdivision can get expensive.

4. Vector averaging across rows is not supported because this averaging is done on-the-fly at plot-time, after the TaQL expressions have been evaluated. The TaQL expressions force the data to be made Scalar, to conform to the data storage arrays. One solution is pre-averaged data sets, which can be done via `VisibilityIterator`.

5. Iteration plots only move forward, following the `casa::TableIterator` functions.

5.2 Making changes

1. **Different data source** : Need to create a class that inherits from `casa::BasePlot`, and either (a) fills in the `casa::Array`s currently used to hold data, or (b) provides a set of query functions that `casa::TPPlotter` can use to access the data and flags to be plotted. (a) will allow the existing Flag/Locate user interaction to work without intervention, and will require functions that translate the final flags when writing back to disk. (b) will allow data storage in a different format (perhaps amenable to memory mapping for the plotting package) but will require a `casa::BasePlot` level implementation of Flag/Locate operations. The rest of the system should not need to know about the data access mechanism internal to `casa::BasePlot`. The `casa::TablePlot::createBP` function decides which version of `casa::BasePlot` to use for a particular Table. Adding functionality this way, allows different types of Tables and access mechanisms to be used together to create active plots of different kinds and have them appear simultaneously on the plot window.

2. **Different plotting package** : Based on the above implementations of data access and plotting operations and common functionality requests from users, the following is a (minimal) list of requirements from any prospective plotting package.

   (a) Draw scatter plots, lines, histograms and render them fast (maybe raster).

   (b) Handle time formatting (and co-ordinates) for axis tick labels

   (c) Configure multiple panels and handle overlays

   (d) Draw rubber-band boxes and return region coordinates

   (e) Customizable GUI for adding buttons.

   (f) Allow memory mapping between `casa::Array` and the plotting package data format

   (g) Must not hold an extra copy of all the data being plotted

   (h) Allow some way of accessing the plotter GUI from the `Casapy` command line.

   Within the C++ code, one will need to replace a well defined set of plotting-package specific functions in `casa::TPPlotter`. Currently, for Matplotlib, they fill in `PyArrayObject` arrays and construct command strings for various functions. These functions were originally written for PGPLOT, and were later modified for PLPLOT and now Matplotlib.

3. **Different Plot Style (histogram/3D plots)** : The functions that construct the plot commands can be augmented to generate histograms of data that is read in via existing data access mechanisms.

6 Appendix A : Sample TaQL expressions

The TaQL [3] syntax is used to construct arithmetic expressions using Table column names to compute values to be plotted. Any TaQL expression resulting in an Integer or Double/Float Scalar or Array can be used. Expressions resulting in Complex or Boolean values are invalid. Scalar and vector reduction functions are supported.

1. Sample expressions for a Measurement Set are shown below. Channel and correlation selection is done via one-based TaQL indices.

   ```
   uv distance :  SQRT(SUMSQUARE(UVW[1:2]))
   amp of data :  AMPLITUDE(DATA[1:2,1:10:1])
   amp of data averaged over chan/corr :  MEAN(AMPLITUDE(DATA[1:2,1:10:1]))
   amp of data averaged over chan only :  MEANS(AMPLITUDE(DATA[1:2,1:10:1]),2)
   ```

2. For 2D X-Y plots separate TaQL strings need to be supplied for the X and Y axes. The resultant data shapes for X:Y must be either 1:1 or 1:N or N:N.

   ```
   1:N : 'SQRT(SUMSQUARE(UVW[1:2]))',AMPLITUDE(DATA[1:2,1:10:1])'
   1:1 :  'SQRT(SUMSQUARE(UVW[1:2]))',MEAN(AMPLITUDE(DATA[1:2,1:10:1]))'
   1:1 :  'TIME/86400.0+678576.0','AMPLITUDE(GAIN[1,5])'
   N:N : 'REAL(DATA[1:2,1:10:1])','IMAG(DATA[1:2,1:10:1])'
   ```

3. For derived quantities that cannot be expressed as TaQL strings, a C++ interface has been provided to specify any arbitrary conversion function. This conversion function will be applied to the data extracted out via the TaQL queries.

   ```
   Hourangle vs time :
   TaQL X,Y pair :  'TIME/86400.0+678576.0','TIME'
   X conversion function :  NULL
   Y conversion function :  Use the raw MJD TIME values, the Measures classes
   and field-centre locations to compute HA.
   ```

4. Values from an ArrayColumn can be plotted against cell row or cell column indices (implemented in `casa::CrossPlot`). For a measurement set, this allows plots of " Amp(Data) Vs Frequency or Channel ". TaQL X-Y pairs for such plots need explicit specification for only Y-axis values. Zero based channel indices will automatically be placed on the X axis. If frequency values are required, a C++ conversion function can be applied to the channel indices.

   ```
   TaQL X,Y pair for Amp vs Channel:  'CROSS','AMPLITUDE(DATA[1:2,1:10])'
   TaQL X,Y pair for Amp vs Frequency:  'CROSS','AMPLITUDE(DATA[1:2,1:10:1])'
   X conversion function :  Use channel numbers to index into the SPECTRAL_WINDOW
   subtable and read frequency values.
   Y conversion function :  NULL
   ```

[3] Aips++ Note 199 : *http : //aips2.nrao.edu/stable/docs/notes/*199/199*.html*

A plot option "columnsxaxis=True/False" controls whether column or row indices of the Array are plotted on the X-axis.

5. **Scalar averaging within a Table row**

   Averaging across cells in an ArrayColumn can be done via the TaQL reduction functions MEAN and MEANS.

   ```
   Ignoring Flags :  'MEAN(AMPLITUDE(DATA[1:2,1:10]))'
   ```

   Weighted averaging across cells in an ArrayColumn can be done via the TaQL reduction functions SUM and SUMS.

   ```
   Honouring Flags :
   Weighted averaging with FLAG
   'SUM(AMPLITUDE(IIF(FLAG[1,1:10],0.0,DATA[1,1:10]))) /SUM(IIF(FLAG[1,1:10],0.0,1.0))'

   Weighted averaging with WEIGHT_SPECTRUM
   'SUM(AMPLITUDE(DATA[1,1:10])*WEIGHT_SPECTRUM[1,1:10]) /SUM(WEIGHT_SPECTRUM[1,1:10])'

   Weighted averaging with IMAGING_WEIGHT
   'SUM(AMPLITUDE(DATA[1,1:10])*ARRAY(IMAGING_WEIGHT[1:10],[1,10]))
        /SUM(ARRAY(IMAGING_WEIGHT[1:10],[1,10]))'

   Weighted averaging with WEIGHT
   'SUM(AMPLITUDE(DATA[1,1:10])*ARRAY(WEIGHT[1],[1,10]))
        /SUM(ARRAY(WEIGHT[1],[1,10]))'

   Weighted averaging with FLAG and IMAGING_WEIGHT
   'SUM(AMPLITUDE(IIF(FLAG[1,1:10],0.0,DATA[1,1:10]) *ARRAY(IMAGING_WEIGHT[1:10],[1,10])))
        /SUM(IIF(FLAG[1,1:10],0.0,1.0) *ARRAY(IMAGING_WEIGHT[1:10],[1,10]))'
   ```

6. **Vector averaging within a Table row.** The vector reduction functions need to be applied to the complex values before conversion to a scalar. `Ignoring flags :`

   ```
   'AMPLITUDE(SUM(DATA[1,1:10]))'
   Honouring flags :  'AMPLITUDE(SUM(IIF(FLAG[1,1:10],0.0,DATA[1,1:10]))
                      /SUM(IIF(FLAG[1,1:10],0.0,1.0)))'
   ```

7. **Scalar averaging across Table rows.** TaQL expressions cannot be written for reduction operations that span multiple rows of the Table. Therefore, the `averagenrows` plot option can be used to average the result of the Y-TaQL expression across rows. Flagged points are automatically ignored.

8. **Vector averaging across Table rows** : Not supported, because averaging across Table rows is done after TaQL evaluations that force the values to be made scalar before averaging. See the section on Known Problems for details.

7 Appendix B : List of Plot Options

1. **nrows, ncols, panelindex** (default = 1,1,1) : The Matplotlib syntax is used to choose panel locations on the plot window. $1 <= panelindex <= nrows \times ncols$

2. **plotrange = xmin,xmax,ymin,ymax** (default = all) : The range of data to be plotted. When the range chooses a subset of the data, points are plotted from those stored in memory.

3. **plotrangesset = xminSet,xmaxSet,yminSet,ymaxSet** (default = none) : A bitmask to indicate which data ranges to honour. To be used with **plotrange**.

4. **timeplot** (default = 'o') : To set time-formatting for tick labels. Matplotlib times are referenced to 01/01/0001. Therefore, the conversion $TIME/86400.0 + 678576.0$ must be applied to the MJD values stored in the Table. Options are 'x':xaxis,'y':yaxis,'b':both,'o':off to choose which axes to apply time formatting.

5. **columnsxaxis** (default = True) : Plots of data from only one ArrayColumn (Y-TaQL) use X-TaQL as 'CROSS' and use either column or row indices of the Array cell for the x-axis. For a Measurement Set to get channel numbers on the x-axis, set this parameter to True. False will place correlation indices on the x-axis.

6. **overplot** (default = False) : If True, make a new plot layer and place on top of the current stack of plots. If False, all existing plots on the current panel are cleared before drawing the new plot.

7. **replacetopplot** (default = False) : To be used along with 'overplot=True'. This replaces only the top plot layer, but keeps all lower layers intact. (For example : after making an overplot, the plotsymbol for the top-most layer is to be changed without having to replot all existing layers)

8. **removeoldpanels** (default = True) : If True, follow Matplotlib convention of clearing away plot panels if a new panel is about to overlap it. If False, all new panels will be placed on top of existing ones even if they overlap. An explicit 'clearplot' is required to clear panels.

9. **fontsize** (default = 12) : Matplotlib font size for the title. X and Y labels and tick labels are 80% of this size.

10. **xlabel, ylabel, title** (default = none) : Label strings for the x-axis, y-axis and the title. Multi-line labels need
    n for every newline.

11. **windowsize** (default = 8.0) : GUI window width in cm (matplotlib convention)

12. **aspectratio** (default = 1.0) : Window height/width. Must be the same for all panels.

13. **doscalingcorrection** (default = False) : Try not to use this. To be set to True only if the "MEAN" Array reduction function is used along with FLAGs or WEIGHTs. It rescales the average to account for the fraction of data points not used in the MEAN.

14. **separateiter** (default = 'none') : To choose panel arrangement when multiple iteration plots are setup for simultaneous iteration. 'none' : multiple iteration plots run in overplot mode. 'row/col' : multiple iteration plots run in different rows/columns of panels.

15. **honourxflags** (default = False) : To be used only with "X-TaQL=CROSS" mode with channel averaging, to decide how the average channel number is to be computed. False : Compute the average x axis value as the middle of the range being averaged. True : Compute the average x axis value accounting for flagged cell rows/cols.

16. **locatecolumns** (default = none) : This is the list of Table columns that will get sent into the "locate" function when triggered by the GUI. It will be overridden by anything specified in a DumpLocateInfoBase class.

17. **plotsymbol** (default = ',') : Matplotlib plot symbols

18. **color** (default=(1.0,0.4,0.2)) : Matplotlib colour string. Can be a predefined pylab colour 'brown', or an html hex string '#7FFF4e', or '(r,g,b)'. If specified (length¿0), this takes precedence over the colour specified via PlotSymbol. If a non-predefined colour is specified, multicolour is always False.

19. **pointlabels** (default = none) : A vector of text labels to be applied to the first N data points. Flagged points and their labels will not appear.

20. **markersize** (default = 10.0) : Matplotlib point marker size.

21. **linewidth** (default = 2.0) : Matplotlib line-width.

22. **multicolour** (default = 'none') : Optional multi-colouring of rows and columns in an ArrayColumn. 'cellrow' : Array cell rows get different colours, 'cellcol' : Cell cols get different colours, 'both': rows and cols get different colours,'none': rows and cols get the same colour.

23. **tablemulticolour** (default = True) : When multiple Tables are sent in simultaneously into TablePlot, plot them in different colours.

24. **showflags** (default=False) : If True, plot only flagged points in the reserved colour 'magenta'.

25. **flagversion** (default='main') : Use flags from a specified flag-version while displaying the data. Flag edits will write back to this flag version table.

26. **skipnrows** (default=1) : Start with the first point, and then plot only if npoints % SkipNRows == 0. When plotting unflagged data, flagged points are not counted while skipping.

27. **averagenrows** (default=1) : If larger than 1, average together every N rows. ArrayColumn cell rows and columns are kept distinct. Averaging across ArrayColumn cell rows/cols can be done directly via TaQL reduction functions.

28. **connect** (default='none') : Draw lines through data points along a specified axis. 'tablerow' connects points across rows. 'cellcol' connects points across ArrayColumn cell columns (channels). 'cellrow' connects points across ArrayColumn cell rows (correlations).