1. Introduction

This document is supposed to be an overview document and describes how to use the AthenaTools Plotter Widget Set to add graphs to your X programs. It is assumed the reader is familiar in general with programming X and widget sets. All of the detailed information regarding resources etc. is documented in the relevant manual pages.

The AthenaTools Plotter Widget Set provides many of the features scientific and business users need in a graph that will be embedded in another program. Some highlights of the plotter widgets are

- X-Y plots for scientific-style graphics.
- Line and bar plots for business-style graphics.
- Automatic or manual scaling of axis values, automatic selection of tic and subtic points.
- Callbacks to get the current pointer position in the plotting area in pixel and user coordinates, and to cut off a range or rectangle from the plotting area for zooming in, for example.
- Very flexible text facilities, including mathematical symbols, sub- and super-scripts, font changes etc. These text facilities can be used for legend names, graph titles, axis labels, tic labels, or attached to arbitrary points on the graph in either pixel space or user space.
- Unlimited number of plots on each graph, with each plot independently specified[1].
- Encapsulated PostScript generated that closely matches the screen output.
- Logarithmic and linear axes, or axes labeled at arbitrary points with arbitrary strings (which is very useful for labeling axes with dates, for example).
- Very flexible axis handling. There are up to four axes on the graph, and any plot can be associated with either X axis and either Y axis.
- "Frame" axes that get their scaling etc. from another axis object, on either the same or another graph. This makes coordination between related graphs very easy.
- A Fast Update mode that can quickly redraw rapidly-changing data[2].
- Easy access to application data. The widget set is designed to access application data that is stored in an array of structures; it can pull data out of structures in either Float, Double or Integer format, but other data types are easily added.

The Plotter Widget is a widget based on Xlib and Xtoolkit. It is intended to be used in conjunction with other widget sets based on Xlib and Xtoolkit, like the Athena Widget Set or the OSF/Motif Widget Set. The Plotter Widget Set is not intended for stand-alone graphic applications (use GNUplot or similar for that). It is not a 3D library, and is not really a replacement for Harvard Graphics or some such. It cannot do Pie charts or other non-XY type graphs.

[1] Each plot on the graph is a different widget.

[2] This is implemented but untested at this point in time.

2. Usage Walkthrough

This section gives a quick overview of how to program using the AthenaTools Plotter Widget Set, and explains what sort of data the plotter expects from your application. There can be found four test programs using the Athena Widget Set in the subdirectory `athena`, and one larger test program using the OSF/Motif Widget Set in the subdirectory `motif`. You can make and run these programs by typing `xmkmf` to create the `Makefiles` from the `Imakefiles`, and then typing `make all` or `make` to create the executables. All these programs are written in C and use no resource files to run properly. This means anyone should be able to compile and run them. You should be able to transfer the concepts to your favourite development environment with no problems. The example source code in this section is taken from two test programs that came with the source. The file `Plot.c`, containing all the stuff how to use and program the plotter widgets, from the Motif based example program `mplot`, and the file `sample_prog.c`, a Athena based application, are completely listed in Section 5. The title page of this document, and the Figure 1 below, a plotter PostScript output from `mplot`, show a typical scientific use of the plotter widget set.

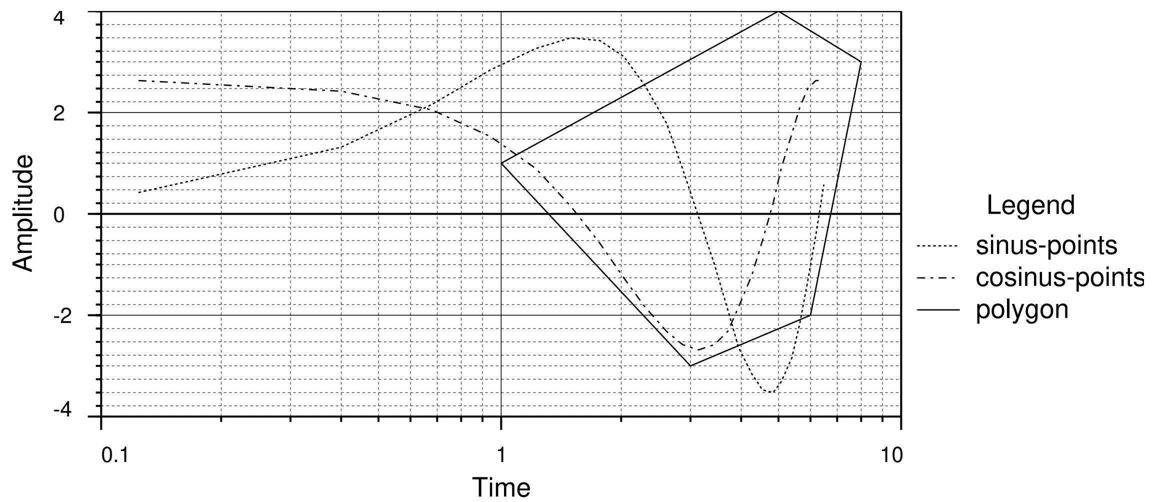

Figure 1: PostScript Output From The `mplot` Test Program

2.1. Creating The Plotter And The Axes

The general arrangement is as follows. For each graph, you create a single AtPlotter widget. This is a widget, and has an X window, the same as most other widgets. You then create some axis objects (some subclass of AtAxisCore, usually AtXYAxis or AtAxis) as children of the AtPlotter and attach them to the parent by doing an `XtSetValues()` using the `XtNxAxis` and `XtNyAxis` resources. In the sample program `mplot`, a more scientific use of the plotter widgets, this is done in the procedure `MakePlotter()` which is called when the application shell, the main window, some buttons and other widgets, and a parent form and frame for the plotter are created:

```
#include <At/Plotter.h>
#include <At/XYAxis.h>

Widget parent;          /* Already created! */
Widget plotter, xaxis, yaxis;
Arg args[8];
int n;

/* Create the plotter */
n = 0;
XtSetArg(args[n], XtNwidth, 400); n++;
XtSetArg(args[n], XtNheight, 300); n++;
XtSetArg(args[n], XtNborderWidth, 0); n++;
XtSetArg(args[n], XtNshowLegend, True); n++;
plotter = XtCreateManagedWidget("plotter", atPlotterWidgetClass,
                           parent_widget, args, n);

/* Create the X axis */
n = 0;
XtSetArg(args[n], XtNlabel, "Time"); n++;
XtSetArg(args[n], XtNaxisWidth, 1); n++;
xaxis = XtCreateWidget("xaxis", atXYAxisWidgetClass, plotter, args, n);

/* Create the Y axis */
n = 0;
XtSetArg(args[n], XtNlabel, "Amplitude"); n++;
XtSetArg(args[n], XtNaxisWidth, 1); n++;
XtSetArg(args[n], XtNvertical, True); n++;
yaxis = XtCreateWidget("yaxis", atXYAxisWidgetClass, plotter, args, n);

/* Attach X and Y axes to the plotter */
n = 0;
XtSetArg(args[n], XtNxAxis, xaxis); n++;
XtSetArg(args[n], XtNyAxis, yaxis); n++;
XtSetValues(plotter, args, n);
```

This is the minimal configuration for graphs using the AthenaToolkit Plotter Widget Set[3].

2.2. Creating Plot Objects And Attaching Data

Then, for each data item to plot on this graph, you create a plot object, in this example AtXYLinePlot, a subclass of AtXYPlot:

```
#include <At/XYLinePlot.h>

Widget plotter;         /* Already created! */
Widget line1;
Arg args[4];
int n;

/* Create line 1 */
n = 0;
XtSetArg(args[n], XtNlegendName, "sinus-points"); n++;
XtSetArg(args[n], XtNplotLineStyle, AtLineDOTTED); n++;
XtSetArg(args[n], XtNplotMarkType,  AtMarkRECTANGLE); n++;
line1 = XtCreateWidget("line1", atXYLinePlotWidgetClass,
                    plotter, args, n);
```

[3] A complete listing of the source file `Plot.c` from which the examples are taken, is enclosed in Section 5.1.

When the plot objects are created, you can use the member functions for that plot object to attach it to your application data. In the test program the data, a simple polygon and some curves, are created from within the routine `MakeDemoData()`, or are read in from a data file in the procedure `ReadPlotData()`. Once the data are attached and the AtPlotter is first displayed, it will automatically calculate all the bounds, scaling etc. and display the graph.

In contrast to previous releases of this widget set[4], copies of the data are not required. The assumption is your application data are stored in an array of structures, statically declared as in `MakeDemoData()` or dynamically allocated as in `ReadPlotData()`. For the AtXYLinePlot class there are three pre-defined data types `AtDoublePoint`, `AtFloatPoint` and `AtIntPoint` which correspond to the currently supported data types `AtDouble`, `AtFloat` and `AtInt`. These are simple structs containing one item for the X coordinate and one item for the Y coordinate of a data point. `AtDoublePoint`, for example, is defined in `At/XYLineplot.h` as follows:

```
typedef struct {
    double x;           /* X coordinate */
    double y;           /* Y coordinate */
} AtDoublePoint;
```

An array of such a struct type is just a line composed of its data points. For the polygon in Figure 1, a connected line defined of 6 points, you would declare an array of 6 data points:

```
#include <At/XYLinePlot.h>

#define PNUM 6
static AtDoublePoint polygon[PNUM];
```

and attach the polygon to a plot object. using the member routine `AtXYLinePlotAttachDoublePoints()`[5]. In the example this is done in the procedure `AttachData()`:

```
Widget line3;           /* Already created! */

/* Attach polygon from predefined type AtDoublePoint to line3 */
AtXYLinePlotAttachDoublePoints(line3, &polygon[0], 1, PNUM);
```

The parameters for this routine are :

- The widget `line3` (a subclass of AtXYPlot) you are attaching the data to.
- A pointer `&polygon[0]` to the first item in the static array.
- The index of the first item in the array which should be plotted (often 1 but not always).
- The number of items `PNUM` in the array which should be plotted.

The data pointed to in this routine should remain valid until the next attach call is made (as it may be accessed if the window resizes or if other plots change or if PostScript output is generated). Setting the address to NULL and the count to 0 will effectively undisplay the plot. If the underlying data has changed, a new call to the attach routine should be made so that the

[4] Up to Version 4.

[5] This routine is a `#define`'d alias for the routine `AtXYPlotAttachData`.

change is noted and new bounds are calculated.

In addition, providing your application is suitably arranged, the plot widgets can read the information straight out of your existing structures. Assume you have defined a record type, composed of one common X value and two Y values, like the `struct point`:

```
struct point {
    double xval;        /* The X value  */
    double yval1;       /* The Y values */
    double yval2;
};
```

and you have 24 items of that type. You would then declare an array of 24 records:

```
#define CNUM 24
static struct point curve[CNUM];
```

and use the appropriate member routine `AtXYLinePlotAttachData()`[6] to attach the data. This routine is more general than the above denominated member routine `AtXYLinePlotAttachDoublePoints()` and somewhat more complicated:

```
/* Attach curve from own defined type point to line1 */
AtXYLinePlotAttachData(line1,
    (XtPointer) &curve[0].xval,  AtDouble, sizeof(struct point),
    (XtPointer) &curve[0].yval1, AtDouble, sizeof(struct point),
    1, CNUM);
```

The parameters are as follows:

- The widget `line1` (a subclass of AtXYPlot) you are attaching the data to.
- A pointer `&curve[0].xval` to the first X value data item in the array.
- The type `AtDouble` of that item.
- A parameter `sizeof(struct point)` called the Stride, which is the number of bytes between successive data items. In most applications, that will be the size of the structure you are accessing.
- A pointer `&curve[0].yval1` to the first Y value data item in the array.
- The type `AtDouble` of that item.
- The stride parameter `sizeof(struct point).`
- The index of the first item in the array.
- The number of items in the array which shoud be plotted.

Once the `XtMainLoop()` routine is called after creating the plotter, the axes and the lines, and after attaching the data to the lines, the graph magically appears.

2.3. Using The LinePlot And BarPlot Classes

As an example for a more business-like graphic using line and bar plots and two Y axes, consider the second test program `sample_prog`, a simple accounting application for a Widget

[6] This routine also is just a `#define`'d alias for the routine `AtXYPlotAttachData()`.

Distribution Company[7]. You might wish to display the weekly sales and profit figures for the sale of Widgets for the past year. But your database stores only the volume sold, the sale price and the purchase price. So you would declare a structure like this:

```
struct profits {
    /* From database */
    long volume;
    float buy_price;
    float sell_price;
    /* Calculated values */
    double sales;
    double cost;
    double profit;
};
```

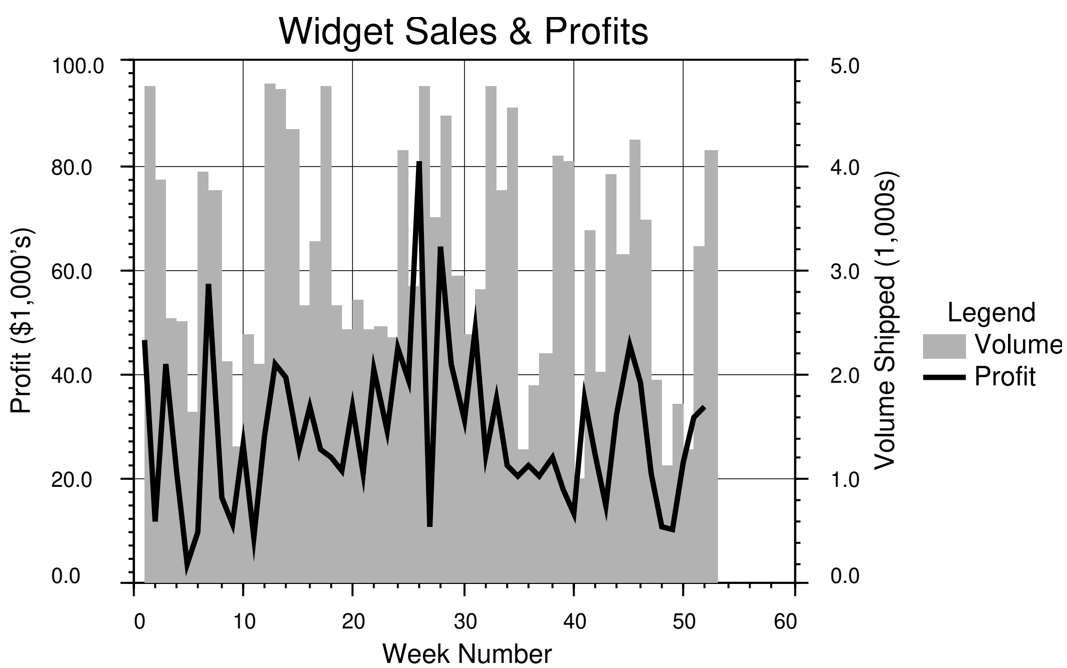

Figure 2: PostScript Output From The `sample_prog` Program

You would then declare an array of 52 structs:

```
#define NUM_WEEKS 52
struct profits profits[NUM_WEEKS];
```

Two routines (called `ReadFromDatabase()` and `CalculateProfits()` in the sample program) could read in the weekly figures from the database and run through a loop to calculate sales, cost and profits for each week. This creates the data to be plotted. Like in the first

[7]The complete listing of `sample_prog.c` is enclosed in Section 5.2.

example you have to create the plotter and its children, when application shell and parent form are created. The routine `MakePlotter()` does that: it creates the main plotter, then the X axis for the week number, the Y axis for profit, and the Y2 axis (on the right hand side) for volume. Then it creates the plot objects, one AtLinePlot object for profits and one AtBarPlot object for volume.

The routine `AttachData()` attaches the calculated data to the plot objects using the member routine `AtLinePlotAttachData()`[8], with a call like:

```
AtLinePlotAttachData(line, (XtPointer) &profits[0].profit,
                     AtDouble, sizeof(struct profits), 1, NUM_WEEKS);
```

The parameters, similar to that in the first example above, are as follows: The widget `line` (a subclass of AtSPlot) you are attaching the data to, a pointer `&profits[0].profile` to the first data item, the type `AtDouble` of that item, the stride `sizeof(struct profits)`, the integer X axis value of the first item in the list, and the number `NUM_WEEKS` of items in the list. The X axis value of the last item will be (start + num - 1).

The volume data is attached to the AtBarPlot widget with the equivalent call[9] :

```
AtBarPlotAttachData(bar, (XtPointer) &results[0].volume,
                    AtInt, sizeof (struct profits), 0, NUM_WEEKS);
```

with the same parameters as above.

2.4. Differences Between The AtXYLinePlot And AtLinePlot Classes

Lets close this section by pointing out the main differences between the AtXYLinePlot and the AtLinePlot class, as can be seen in the examples given in the above subsections:

- When attaching data to an object of the AtXYLinePlot class you have to specify explicitely the X and the Y values of your data, no matter if you use one of the pre-defined structure types like `AtDoublePoint` or specify your own data types. As you would expect you can define any range for the X and the Y values, as it is needed for most scientific applications, including connected lines like the polygon in the first example (see Section 2.1).
- In contrast to an AtXYLinePlot object objects from the AtLineplot or AtBarPlot classes require data with a continuous X range. The X values are taken as integers from the indices in the data arrays. You can set the `XtNticMultiplier` resource to make the X indices and the tic labels more conform to your real X data, but it's impossible to specify a range from xmin = -3 to xmax = +3, for example, or to plot connected lines.

[8] This routine is alos just a `#define`'d alias for the routine `AtSPlotAttachData()`.

[9] The `AtBarPlotAttachData` is also a `#define`'d alias for `AtSPlotAttachData`.

3. More Detail

3.1. Definitions

Lets start by defining some terms so we all understand what we are talking about.

- **Object**
  Any X Toolkit Intrinsics object that is a subclass of the Object class. `XtSetValues` operates on Objects and `XtCreateWidget()` creates them.
- **Widget**
  Any Object that is a subclass of Core and thus has a window.
- **Graph**
  The entire display of a Plotter widget which includes the main plotter widget and all children, axes and plots. There can be multiple graphs, and hence multiple AtPlotter widgets, in one program.
- **Plot**
  A single data Object plotted on a graph. (i.e. a single line or bar plot). One AtPlot Object per plot, multiple plots per graph.
- **User Space, Pixel Space**
  User space is the realm of real numbers that the graph is attempting to plot. Pixel space is the screen coordinates that X windows works in. All application data is expressed in user space, and is converted to pixel space (or pixel locations) by the various recalc routines inside the plotter. Application programmers should deal with user space only, and not need to touch pixel space.
- **Refresh**
  Simply redraw one plot on the graph. No screen clear or redraw of other child Objects required.
- **Redraw**
  Simply refresh the on screen image of the whole plotter widget. No calculation required.
- **Relayout**
  The pixel dimensions of the graph widget have changed (e.g. because the user resized the application) or the size of one of the component parts (e.g. an axis is wider because the font changed), but the data that is plotted is unchanged.
- **Rescale**
  The data has changed and the existing axis bounds are not sufficient. New axis bounds are required, but no layout has changed and all axes are in the same spot. Is concerned with min/max of axis values, tic values and intervals etc.
- **Recalc**
  Either pixel location or data has changed for a plot, and the cached pixel values need to be recalculated. Can be for a single child or the whole graph.

A rescale implies a recalc, and a relayout implies a recalc, but relayout and rescale are orthogonal (i.e. one doesn't necessarily imply the other). A recalc implies a redraw.

3.2. Widget Class Hierarchy

Figure 3 below shows the class inheritance of the AthenaTools Plotter Widget Set:

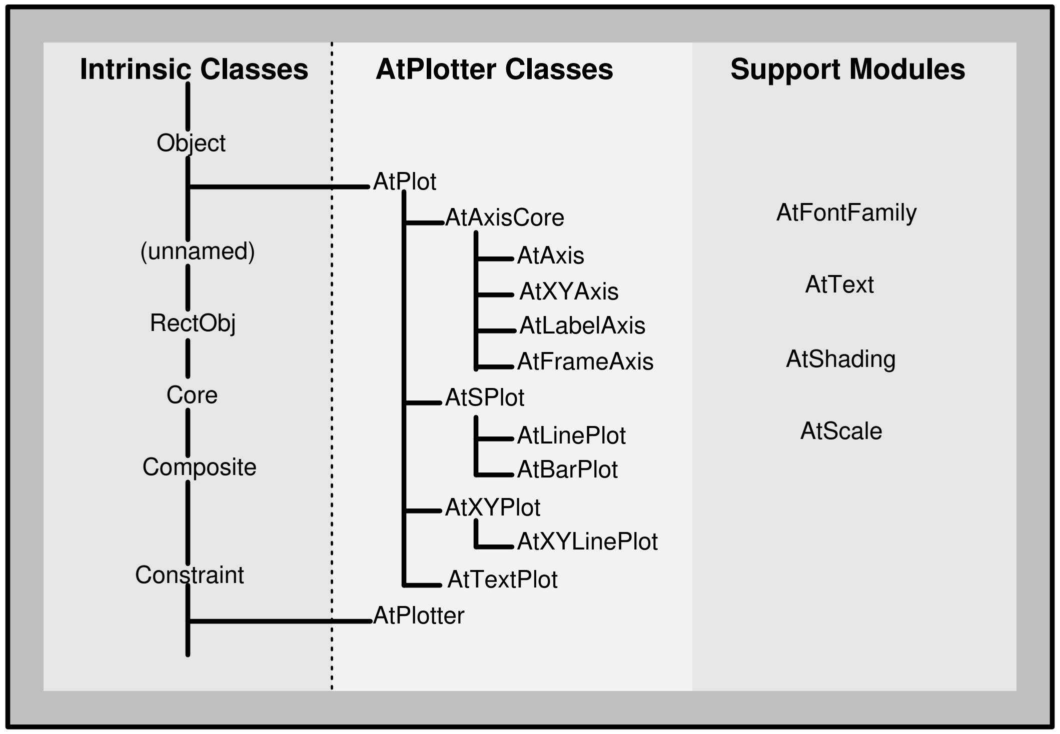

Figure 3: The Plotter Widget Set Class Hierarchy

As can be seen, most of the widgets in the set are actually Objects and don't have an X window of their own. They draw on the the window supplied by the parent AtPlotter widget. A minimal graph includes a single AtPlotter widget that is the parent of two Objects that are subclasses of AtAxisCore, and one Object that is a subclass of AtSPlot or AtXYPlot.

The AtPlot, AtSPlot, AtXYPlot and AtAxisCore widget classes are simply placeholders for some logic and resources; you shouldn't ever need to make Objects of these types. The AtPlotter will only accept children that are a subclass of AtPlot, and AtPlot classes will only accept AtPlotter widgets as a parent. We will examine each of the elements in turn.

3.3. Elements of the Graph

Figure 4 on the next page shows the various areas of the graph. The string along the top is known as the title. The font, size, color and style (i.e. roman, *italic*, **bold** etc.) and the string itself can be set using resources of the AtPlotter widget. The font family (see Section 4.1) is the same for all the strings in the AtPlotter widget. Specifying a title as the empty string removes the title and reclaims the space for the main part of the graph.

Along the right hand edge of the graph is the legend. The AtPlotter maintains details about the legend as Constraint resources on behalf of the children. The AtPlotter has resources to set the string, size and style of the legend title, as well as the size and style of the legend labels.

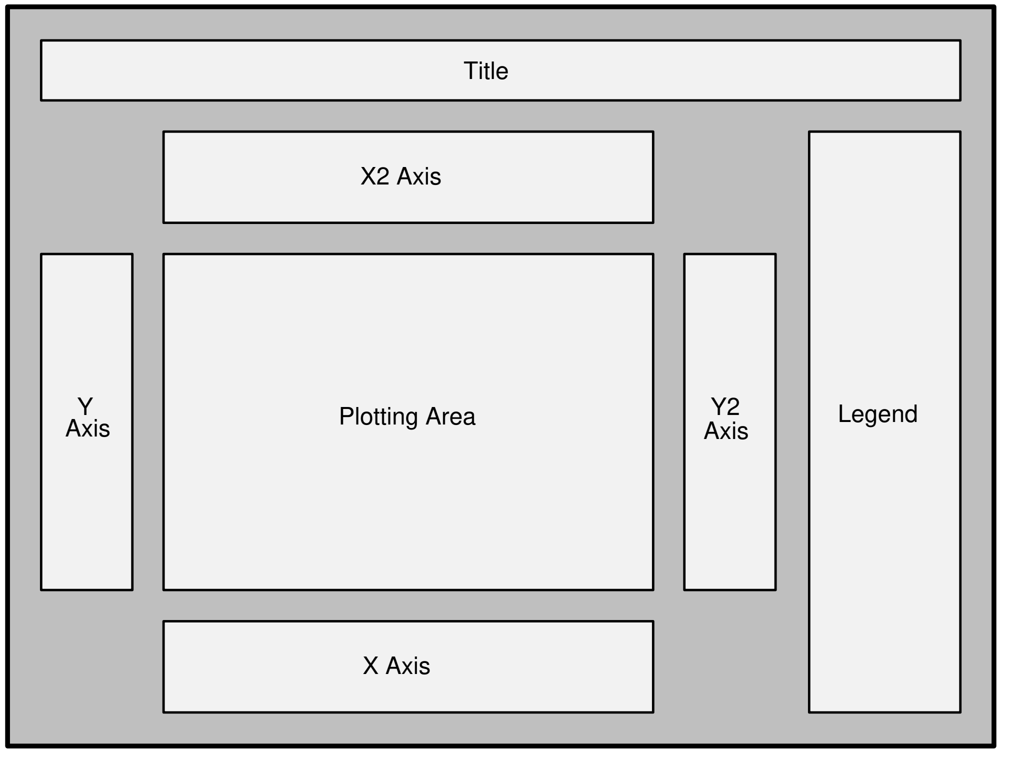

Figure 4: Elements Of The Plotter Window

The color of the text in the legend is set in the AtPlotter and affects all the text in the legend. The actual legend strings are constraint resources and belong (logically) to the child. Each item (apart from the title) in the legend is drawn by the child plot, and exactly matches the corresponding plot in the main graph area. The legend is controlled by the AtPlotter widget, and the resource `XtNdrawLegend` controls whether it is displayed.

There are two axes used in the sample graph shown in Figure 1, the horizontal axis is a logarithmic axis of class AtXYAxis, the vertical axis is a linear axis, also of class AtXYAxis. All the values displayed on these axes were calculated automatically (although you can manually set the maximum, minimum and the distance between tic marks if required). There are 3 tics on the logarithmic X axis, and 5 tics on the linear Y axis. Each tic has a label and a tight dotted grid line associated with it. The color of the axis, the font, size and style of the tic labels and a `sprintf()`-like format string can be specified.

The small marks between the tic values are called subtics, and have a wider dotted optional subgrid line associated with it. They are automatically generated, whether the main ticks are manually or automatically set. For the AtAxis class the subtic density resource can increase or decrease the number of subtics per tic, and care is taken that the number is sensible (e.g. power of 2 or 5 etc).

There is a label associated with each axis, again an arbitrary text string in any font, color or style. For vertical axes it is rotated 90 degrees. These are the labels "Time" and "Amplitude" in the sample graph.

In general, there are four axes on the graph. The horizontal axes are the X axes, the lower is known as the X axis and the upper as the X2 axis. The vertical axes are Y axes, the left being Y axis and the right being Y2 axis. Each plot on the graph is associated with one X axis and one Y axis. The AtPlotter maintains constraint resources on behalf of each plot child to decide which X and Y axis it is associated with. If there are no plot children associated with an axis (as is the case for the Y2 and X2 axes in the graph shown in Figure 1 or for the X2 axis in the graph shown in Figure 2), then there is no need for an axis widget to be created for that axis.

The X2 and Y2 axes are known as mirror axes, and the AtAxisCore resource `XtNmirror` **must** be set on axes used for these axes and **must not** be set on axes used for the X axis and Y axis. Similarly, the AtAxisCore resource `XtNvertical` **must** be set on axes used for the Y axis and Y2 axis, and **must not** be set on axes used for the X axis and X2 axis. Both `XtNmirror` and `XtNvertical` resources can be set at Object creation time only; any attempt to change them will fail with a warning message.

The AtAxisCore widget is a placeholder for most of the axis resources, and also contains the logic for calculating pixel values from user-space values. It relies on subclass member functions to decide what values to use for minimum and maximum, at what values to place tic and subtic marks, and the labels associated with those tic marks. The AtAxisCore widget then handles all the pixel conversions and screen and PostScript redrawing, so the axis classes are very simple to implement.

When the AtPlotter widget is created, it does **not** create any child axes[10].

It is the programmers responsibility to create (at least) two AtAxisCore Objects and attach them to the AtPlotter (using `XtSetValues()` and the `XtNxAxis` and similar resources). Nasty things happen if this isn't done (i.e. the program exits via `XtAppError()`). It **is** possible to have axis children that are not one of the currently used axes, but they must not be displayed (i.e. must have `XtNdisplayed` = False). This is useful for having several possible axes on a plot (e.g. an AtAxis if it is used for scaling a plot, or a AtFrameAxis to carry tic marks when the opposite axis is being used), and to swap between them with calls to `XtSetValues()` using for example the `XtNx2Axis` resource.

Also on the sample graph are three plots from class AtXYLinePlot. For these plots, the color, line type, line style, point marker types and line thickness can be set. This effects both the main plot and the legend entry for that plot. For the AtLinePlot class only the color, line style and line thickness can be set. For the AtBarPlot, the interior of the bars can either be the same color as the outline, or a different color. The interior area can also be stippled with one of 11 stipples[11] that will be converted to a grayscale level on PostScript output. Ether the outline or the interior fill can be disabled (but, obviously, not both).

NOTE: All Objects that are not subclasses of rectObj (and that includes all the AtPlot Objects) do **not** have a managed flag. Hence they should **never** be managed children of the AtPlotter widget. This means they should be created with `XtCreateWidget()` and not `XtCreateManagedWidget()`, and should never be the argument to the `XtManageChild()` routine. Attempts to manage non-rectObjs will corrupt the instance record with undefined results.

[10] This is a change in behaviour since Version 4.

[11] Only 5 distinct on-screen stipple patterns are implemented at the moment, pending discovery of good "10 gray" sort of stipples. The 11 stipples `GRAY0` through `GRAY10` correspond to PostScript `1.0 setgray` down to `0.0 setgray` constructs.

Rather than use `XtManageChild()` and the `XtNmappedWhenManaged` flag to display plots and axes, the parent AtPlotter maintains a constraint variable for each child called `displayed` which may be set with the `XtNdisplayed` resource. Setting this to false has the same effect that calling `XtUnmanageChild()` with the `XtNmappedWhenManaged` flag set would have on a normal widget, i.e. the child would disappear and take no part in the calculation of axes bounds etc. It is an error to undisplay an axis that still has child plots associated with it, and undefined results occur if a normal (as opposed to frame) axis is displayed without children associated with it.

4. Advanced Topics

4.1. Text Facilities

All text strings in the plotter widget set are implemented using the AtText facility. This is in the style of Motif compound XmString type. It is a unified set of routines that allows easy specification of text styles, multiple fonts and sizes as well as symbols, and PostScript output that closely matches the on-screen version. See the `AtText(3X)` manual page for the full details.

Each string segment is made up from five elements: the font family, the style, the size, and the text. The font family is something like "times" and specifies the basic look of the font. The style specifies AtFontPlain, *AtFontITALIC*, **AtFontBOLD** or ***bAtFontBOLDITALIC*** typeface (defined in `At/FontFamily.h`). The size ranges from tiny `AtFontSMALLEST` (about 8 point) to large `AtFontBIGGEST` (about 24 point). Each segment is a simple list of characters in that font/face/size combination. The routine that compiles Strings to the AtText structs has facilities for interpreting escapes to change fonts, to display mathematical symbols etc.

In this widget set, the font family is set once per Object, but the size and style can be changed on a per-string basis. (This avoids the possibility of the so-called "ransom note" look).

4.2. The Layout Procedure

This is the sequence of steps the plotter goes through to lay out a graph.

- The X and Y range for each plot child is accumulated and remembered as data is attached to or changed for each child.
- The X and Y ranges of all the child plots are merged, on the basis of the `XtNuseX2Axis` and `XtNuseY2Axis` resources, into a simple bounding box for all four axes. (One X axis and one Y axis may be unused at this stage).
- **minmax**: Each axis currently in use is in turn offered its minimum and maximum (calculated above) via a call to the AtAxis member function `AtAxisAskRange()`. It is the responsibility of this routine to do any rounding, communication etc. as required, and the returned minimum and maximum are used by the plotter. There is a callback list on the AtAxisCore Object (`XtNrangeCallback`) that is called by the AtAxisCore code after it has decided on the axis bounds. This programmer-written routine can modify the minimum, maximum, number width and tic interval values as required. This is used for linking graphs (see Section 4.7). The calculated number width (which may be approximate only at this stage) is used to calculate the "width" of the axis (which is the amount of the axis that is outside the main graph area).
- A layout is done, assuming the axis widths calculated in step **minmax** are accurate. This sets the position of all the graph components within the main AtPlotter window.
- Each of the active axes is positioned by a call to the `AtAxisSetLocation()` member function. If the length of the axis changes significantly (currently by 10), then it is assumed the number of relevant tics might have changed, thus changing the rounding and endpoints of the axis. If this is the case, the process is repeated at step **minmax**. Should not loop more than twice at this point, assuming step **minmax** can do a reasonable guess for the axis width.

- Each of the axes is recalculated by a call to the AtPlot `AtPlotRecalc()` member function. Most of the arguments to the recalc function for axis objects are ignored. This step may give a better guess as to things like the number width, which changes the layout. If so, the sequence restarts at step **minmax**. Some care is taken to avoid infinite loops.
- Each of the non-axis child plots is recalculated by a call to the `AtPlotRecalc()` routine, with the scales being extracted from the already-recalculated axes as appropriate.
- All children in the graph are redisplayed.

All of this is controlled by one routine which is the redraw routine for the AtPlotter class. All requests are remembered and stored up, so the above procedure is only done when necessary. This means a program can change several things one after the other, and the graph will only do the expensive layout and redraw calculation once, not once per change. This is faster and reduces flickering.

Furthermore, much care is taken internally to avoid as much of the above sequence as is possible, so that if small things change, minimal calculation and redrawing is done.

4.3. Drawing Order

The order in which the children of the AtPlotter widget are drawn affects which is "above" (and hence obscures) which in the display. The first one drawn is conceptually on the bottom, as you would expect. (In this context, "children" refers to all children, both axes and plots). There are three typical situations where this is relevant:

- Generally the grid provided by the axes should be below the plotted data.
- When drawing line plots and bar plots on the same graph, you want the line plots to be drawn second so that they are visible and not obscured by the bars. Otherwise, the information from the line graph will likely be hidden.
- Several bar plots can create the "stacked bar" effect by drawing them in sequence, and arranging them to overlap slightly.

There are two ways the AtPlotter can use to decide which order to draw plots. The first and default method, which is used when the AtPlotter resource `XtNrankChildren` is False, is to draw them in creation order. The first created child is drawn first, and so on. The second method, used when `XtNrankChildren` is True, is to draw them according to a programmer-specified rank order. The parent widget stores a constraint variable called `XtNrank` of behalf of all its children, and will use this to decide which order to draw in. This is more flexible but takes more programmer effort. (When several children have the same rank, they are drawn in unspecified order).

4.4. Customizing Axis Layouts

There are several alternatives to using the default calculated axis bounds and tic positions. The default behaviour (for the AtXYAxis and AtAxis classes) is to pick some number of tics (depending on axis length and the font used to display labels), find a nice, round number for the tic interval that gives approximately that number of tics[12], then round the endpoints up or down to be a multiple of the tic interval. Each tic is given a label by doing a `sprintf()` using the

[12] For AtAxis objects the resource `XtNticDensity` is used in addition to calculate the number of tics per axis.

`XtNticFormat` resource[13]. This is fine is the axis values are numeric, but not very useful if the axis represents dates or categories, or if you need to coordinate axes between separate graphs (for example, having six variables to graph against a common X value, each pair plotted on a separate graph, one graph on top of the other, and all sharing the same X axis values - which would look silly if each X axis was in a different pixel position or had a different scale).

There are three separate mechanisms to address these problems.

4.5. Label Axes

The AtLabelAxis class is very useful if your axis doesn't represent numbers. There are two main areas this could be useful:

- If you are doing something like a stacked bar graph, plotting a couple of figures (say, quarterly sales) for each of several items (say, state by state). Labeling the X axis with numbers is silly. Instead, each X tic can be labeled with a string, i.e. "Vic", "N.S.W." etc. In this situation, you would want to a label and tic for every X axis value.
- If you are plotting values against dates, you would like to be able to specify the axis tics as dates, not numbers. Also, because dates are not regular and months are not all the same length, automatically deciding tic values cannot do the logical thing, such as putting a tic at the start of every month. You would also like to be able to choose which X values have tics and hence labels.

The AtLabelAxis axis allows the application writer to specify arbitrary labels to put on tics, and arbitrary positions to place tics, at the expense of a bit more programming effort. The programmer can specify things like "a tic on the first day of every second month", and label the axes with dates, which cannot be done algorithmically.

The plotter and axis code supports arbitrary X-Y plotting styles, but with exception of the AtXYAxis class and the AtXYPlot class (with the AtXYLinePlot subclass), all other implemented plot widgets assume the X axis is integer valued, and that successive Y values are at successive integral X values (i.e. there are no "holes" in the sequence of Y values). The AtLabelAxis also makes a similar assumption. (This is because the initial use of this widget set is to plot historical data against date, and the AtLabelAxis is used for the X axis.

The application attaches data to the AtLabelAxis with a member function called `AtLabelAxisAttachData()`[14] that acts similarly to the attach procedures for the AtXYLinePlot, AtLinePlot and AtBarPlot classes. It would be used in a manner similar to this:

```
AtLabelAxisAttachData(label_axis, (XtPointer) &answers[0].date_ptr,
                      sizeof(answers[0]), 0, NUM_ANSWERS);
```

The arguments to this function are:

- The AtLabelAxis `label_axis` that you are attaching the data to.
- A pointer `&answers[0].data_ptr` to a String that is the first X axis label.

[13] For AtAxis objects the `XtNticMultiplier` resource is used in addition for labeling the tics.

[14] Which is **not** an alias for the AtSPlot function! This would be one place where multiple inheritance would be nice, as the AtLabelAxis has elements of both AxisCore and AtSPlot classes.

- The stride `sizeof(answers[0]), or number of bytes between successive Strings.`
- The X axis value of the first String. This may or may not be the minimum value for the axis, depending on the `XtNautoScale` resource.
- The number `NUM_ANSWERS` of strings in the array.

If the `n`th String is non-`NULL` and points to a non-empty string, then a tic mark is placed at that X value with the label specified by that string. Otherwise no tic mark or label is placed there. This gives the application writer complete control over labeling the axis. For example, in the above code fragment, the application writer would make sure the `date_ptr` field of the `answers[]` array (which should be a `char *`) would only be set on elements representing the first of the month, and would point to a string like "July, 1990".

4.6. Frame Axes

The AtFrameAxis class is so named because the original use was to put two more axes around simple graphs so that they were completely enclosed. (This gives a more balanced look.) This can also be done setting the `XtNdrawFrame` resource of an object of the AtAxisCore subclasses (like AtXYAxis), which draws a single solid line as axis frame without tics, subtics and labels. An AtFrameAxis object is conceptually attached to another AtAxisCore object (in this or another graph), and, when asked about axis bounds or tic positions, gets the answer by looking at the axis to which it is attached. The frame axis is attached to its target axis by setting the `XtNwidget` resource.

The only things imported from the target axis are the tic and subtic values, the tic labels and the endpoints. The resources concerned with displaying the axis (e.g. things such as color, font, label, tic length, whether or not to draw the tic labels, etc) are individually settable. This is very flexible but may require two axes to be changed instead of one in some circumstances.

One use might be to have a frame axis as the X2 axis attached to the normal X axis. It would get the same tic, subtic and bounds information as the X axis, giving a nice symmetric look. The labels would also be the same, even though they might be rendered in a different font.

4.7. Inter-graph linking

The third method for handling axes is the most flexible but also the most complex for the programmer. As part of the layout processing, each axis gets presented with the minimum and maximum values of all of the plot widgets that are associated with that axis. The axis then does whatever processing is required to determine the endpoints of the axis (in user space, not pixels), such as rounding or examining other axes. Finally, before the values are returned to the main plotter code, the AxisCore code calls the `XtNrangeCallback` callback of the axis. This routine is passed three arguments: the widget ID of the axis, the closure field specified when the callback was added (using `XtAddCallback()` or equivalent), and a pointer to an `AtAxisRangeArgs` structure.

The `AtAxisRangeArgs` structure is defined in `<At/AxisCore.h>` and looks like this:

```
typedef struct {
    double *minp, *maxp, *tic_intervalp;
    int     *max_widthp;
} AtAxisRangeArgs;
```

The four pointers point to the currently chosen values for the maximum and minimum values of the axis, the interval (in user space) between successive tics, and the maximum width (in pixels) of the tic labels. The called routine can chose whatever it likes for these values and, by setting the pointed-to variables, will affect the values used by the axis.

The number width field is supposed to be the width of the widest tic label, and is used to calculate the axis "width" (and hence the pixel position of the axis). Like the axis "width" it represents height of the labels for horizontal axes (as that is what determines the pixel Y position for that axis), and the real width for vertical axes (as that determines the X pixel position). Why is this important? Consider the three stacked graphs discussed above. The pixel position of the Y axis of a graph depends on the width of the Y axis. So to maintain all Y axes at the same pixel locations it is important to make all three Y axes have the same number width (assuming all Y axes have the `XtNdrawNumbers` and `XtNnumbersOutside` resources set to True), as well as making sure the tic length and label resources etc. the same.

4.8. Extending Graphs

A facility that is important in the plot widgets implemented so far is the ability to efficiently handle graphs that are being extended. The intention is to allow efficient and reduced-flicker update of graphs that are being added to in sort-of real time (say once every few seconds). As an example, consider something like xmeter or perfmeter on the Sun - a strip chart that is added to every 20 seconds, and, once every couple of minutes does a shift to the left to leave more room to stuff to be added. The AtSPlot and AtXYPlot classes support this via member functions[15]:

```
AtSPlotDataExtended(widget, num)
```

and

```
AtXYPlotDataExtended(widget ,num)
```

This tells the graph that extra data has arrived, and the new maximum (**not** the number added!) is now `num`. If there is enough room on the X axis, and the new data doesn't go over the Y axis bounds, the new data points are calculated and redrawn with minimum effort, and little or no redrawing of the rest of the display. If either X or Y axis bounds are exceeded, or `num` is less than the current value, the equivalent of an attach is done and the graph is redrawn.

4.9. Callbacks

For the application programmer, the AthenaTools Plotter Widget Set provides additional callbacks[16] to permit user interaction and more flexibility.

Procedures in the `XtNmotionCallback` list are called when mouse motion events occur over the plotting area. That may be used to follow the pointer and to get the current pointer position on the plotting area. The `call_data` argument to the `XtNmotionCallback` is of type `AtPointCallbackData` as defined in `At/Plotter.h`:

[15] There are `for all subclasses of AtSPlot and AtXYPlot, including the AtLinePlot, AtBarPlot and AtXYLinePlot classes.`

[16] The Version 4 provided similar callbacks, but these were dropped out in Version 5.

```
typedef struct {
    short pixelx, pixely;       /* Pixel point coordinates */
    double x1, y1;              /* 1st axes point coordinates */
    double x2, y2;              /* 2nd axes point coordinates */
} AtPointCallbackData;
```

The items `pixelx` and `pixely` contain the pixel coordinates of the current pointer position, `x1` and `y1` the user coordinates scaled against the first X and Y axes, `x2` and `y2` the user coordinates scaled against the second axis X2 and Y2. Some of the example programs make use of this callback.

Procedures of the `XtNclickCallback` list are called when mouse button one is clicked in the plotting aread. That may be used to request the current pointer position in pixel and user coordinates. The `call_data` argument of the `XtNclickCallback` is also of type `AtPointCallbackData` (see above).

Procedures of the `XtNdragCallback` list are called when mouse button one is dragged over the plotting area. This callback may be used to define a rectangle on the plotting area, for example, for further zooming, as it is done by some of the sample programs. The `call_data` argument to the `XtNdragCallback`, also defined in `At/Plotter.h` is of type `AtRectangleCallbackData`:

```
typedef struct {
    short pixelx1, pixely1;     /* Pixel lower left */
    short pixelx2, pixely2;     /* Pixel upper rigth */
    double x11, y11, x12, y12;  /* 1st axes range coordinates */
    double x21, y21, x22, y22;  /* 2nd axes range coordinates */
} AtRectangleCallbackData;
```

`Pixelx1` and `pixely1` contain the pixel coordinates of the lower left corner of the defined rectangle, `Pixelx2` and `pixely2` the pixel coordinates of the upper right corner. In a similar fashion contain `x11`, `y11`, `x12` and `y12` the user coordinates of the lower left corner scaled against the axes X, Y and X2, Y2, `x21`, `y21`, `x22` and `y22` the user coordinates of the upper right corner also scaled against the axes X, Y and X2, Y2.

Care should be taken not to make superfluous use of the `XtNmotionCallback`, `XtNclickCallback` and `XtNdragCallback` callbacks, to avoid unnecessary activity of the plotter widget.

4.10. The Motif Test Program mplot

The OSF/Motif demo program `mplot`, as shown on the title page, should be included in the directory `motif`. Most of the Motif stuff was generated with the User Interface Builder *Motifation*. All the interesting stuff on how to use and program the plotter widgets can bei found in the source file `Plot.c`:

```
/*
 *      Plot.c
 *
 *      Plotter demo program using motif
 *      klin, Mon Aug  3 16:24:02 1992
 */

#include <stdio.h>
#include <math.h>
#include <Xm/Xm.h>

#include <At/Plotter.h>
#include <At/XYAxis.h>
#include <At/XYLinePlot.h>

#include "mplot.xbm"

static Widget plotter;                  /* The plotter widget */
static Widget xaxis, yaxis;             /* The axis widgets */
static Widget line1 = NULL;             /* The 3 demo lines */
static Widget line2 = NULL;
static Widget line3 = NULL;
static Widget line4 = NULL;             /* The 3 user lines */
static Widget line5 = NULL;
static Widget line6 = NULL;

#define TITLE  "Plotter Widget Demo"    /* Plotter title */
#define WIDTH  600                      /* Plotter width */
#define HEIGHT 400                      /* Plotter height */
#define MARGIN 6                        /* Plotter margin */
#define ZOOM   100.0                    /* Max zoom in % */

#define VERSION "0he Athena Tools Plotter Widget Set0ersion 6-beta0atchlevel 3, Aug  4 19920;

#define COPYRIGHT "0 1990 by the Massachusetts Institute of Technology0 1991 by Burdett, Buckeridge & Young Ltd.0 1992 by University of Paderborn0;

#define AUTHOR "0SF/Motif Demo Program by Peter Klingebiel (klin@iat.uni-paderborn.de)0ave fun and enjoy0

char *version = VERSION;
char *copyright = COPYRIGHT;
char *author = AUTHOR;

/*
 *      External widgets used here
 */

extern Boolean display_demo_lines;
extern XtAppContext application_context;
extern Widget appshell;

extern Widget plotter_form, plotter_frame, position_frame, title_label;
extern Widget xpos_label, ypos_label, x_text, y_text;
extern Widget ask_label, zoom_label;
extern Widget linearx_button, lineary_button;
extern Widget logarithmicx_button, logarithmicy_button;
extern Widget subgrid_button;
extern Widget dialog;
```

```
/*
 *      The plot data
 */

struct point {
  double xval;
  double yval1;
  double yval2;
};

#define CNUM 24
#define PNUM 6

static struct point curve[CNUM];

static AtDoublePoint polygon[PNUM];

/*
 *      Create plotter and axes
 */

static void MakePlotter(parent)
  Widget parent;
{
  Arg args[10];
  int n;

  /* Create the plotter */
  n = 0;
  XtSetArg(args[n], XtNwidth, WIDTH); n++;
  XtSetArg(args[n], XtNheight, HEIGHT); n++;
  XtSetArg(args[n], XtNborderWidth, 0); n++;
  XtSetArg(args[n], XtNmarginWidth, MARGIN); n++;
  XtSetArg(args[n], XtNmarginHeight, MARGIN); n++;
  XtSetArg(args[n], XtNtitle, TITLE); n++;
  XtSetArg(args[n], XtNshowTitle, False); n++;
  XtSetArg(args[n], XtNshowLegend, True); n++;
  plotter = XtCreateManagedWidget("plotter", atPlotterWidgetClass, parent, args, n);

  /* Create the X axis */
  n = 0;
  XtSetArg(args[n], XtNlabel, "Time"); n++;
  XtSetArg(args[n], XtNaxisWidth, 1); n++;
  xaxis = XtCreateWidget("xaxis", atXYAxisWidgetClass, plotter, args, n);

  /* Create the Y axis */
  n = 0;
  XtSetArg(args[n], XtNlabel, "Amplitude"); n++;
  XtSetArg(args[n], XtNaxisWidth, 1); n++;
  XtSetArg(args[n], XtNvertical, True); n++;
  yaxis = XtCreateWidget("yaxis", atXYAxisWidgetClass, plotter, args, n);

  /* Attach X and Y axes to the plotter */
  n = 0;
  XtSetArg(args[n], XtNxAxis, xaxis); n++;
  XtSetArg(args[n], XtNyAxis, yaxis); n++;
  XtSetValues(plotter, args, n);
}

/*
 *      Create the 3 lines
 */
void MakeDemoLines()
{
  Arg args[4];
```

```
  int n;

  /* Create line 1 */
  n = 0;
  XtSetArg(args[n], XtNlegendName, "sinus-points"); n++;
  XtSetArg(args[n], XtNplotLineStyle, AtLineDOTTED); n++;
  XtSetArg(args[n], XtNplotMarkType,  AtMarkRECTANGLE); n++;
  line1 = XtCreateWidget("line1", atXYLinePlotWidgetClass, plotter, args, n);

  /* Create line 2 */
  n = 0;
  XtSetArg(args[n], XtNlegendName, "cosinus-points"); n++;
  XtSetArg(args[n], XtNplotLineStyle, AtLineDOTDASHED); n++;
  XtSetArg(args[n], XtNplotMarkType,  AtMarkXMARK); n++;
  line2 = XtCreateWidget("line2", atXYLinePlotWidgetClass, plotter, args, n);

  /* Create line 3 */
  n = 0;
  XtSetArg(args[n], XtNlegendName, "polygon"); n++;
  XtSetArg(args[n], XtNplotMarkType,  AtMarkSTAR); n++;
  line3 = XtCreateWidget("line3", atXYLinePlotWidgetClass, plotter, args, n);
}

/*
 *      Create the plot data and attach them to the demo lines
 */

static void MakeData()
{
  double sin(), cos();
  double x;
  int i;

  /* Create two curves from own data type struct point */
  for(i = 0, x = 0.125; i < CNUM; i++, x += 0.275) {
    curve[i].xval  = x;
    curve[i].yval1 = 3.50 * sin(x);
    curve[i].yval2 = 2.67 * cos(x);
  }

  /* Create a polygon from type AtDoublePoint */
  polygon[0].x = 1.0; polygon[0].y =  1.0;
  polygon[1].x = 5.0; polygon[1].y =  4.0;
  polygon[2].x = 8.0; polygon[2].y =  3.0;
  polygon[3].x = 6.0; polygon[3].y = -2.0;
  polygon[4].x = 3.0; polygon[4].y = -3.0;
  polygon[5].x = 1.0; polygon[5].y =  1.0;
}

static void AttachData()
{
  /* Attach curve from own defined type point to line1 and line 2*/
  AtXYLinePlotAttachData(line1,
    (XtPointer) &curve[0].xval,  AtDouble, sizeof(struct point),
    (XtPointer) &curve[0].yval1, AtDouble, sizeof(struct point),
    1, CNUM);
  AtXYLinePlotAttachData(line2,
    (XtPointer) &curve[0].xval,  AtDouble, sizeof(struct point),
    (XtPointer) &curve[0].yval2, AtDouble, sizeof(struct point),
    1, CNUM);

  /* Attach polygon from predefined type AtDoublePoint to line3 */
  AtXYLinePlotAttachDoublePoints(line3, &polygon[0], 1, PNUM);
}
```

```
/*
 *      Read data from file
 */
static char *ReadPlotData(filename)
  char *filename;
{
  static char msg[256];
  Arg args[8];
  char buf[256];
  AtDoublePoint *data, *p;
  FILE *fp;
  double x, y;
  char *errmsg;
  int nd, n;

  if(fp = fopen(filename, "r")) {
    fgets(buf, sizeof(buf), fp);
    nd = atoi(buf);
    data = (AtDoublePoint *) calloc(nd, sizeof(AtDoublePoint));
    for(p = data, n = 0; n < nd && fgets(buf, sizeof(buf), fp); p++, n++) {
      sscanf(buf, "%lf %lf", &x, &y);
      p->x = x;
      p->y = y;
    }
    fclose(fp);
    if(nd == n) {
      if(line4 == NULL) {
       n = 0;
       XtSetArg(args[n], XtNlegendName,    "test-data1"); n++;
       XtSetArg(args[n], XtNplotLineType,  AtPlotLINEPOINTS); n++;
       XtSetArg(args[n], XtNplotMarkType,  AtMarkDIAMOND); n++;
       XtSetArg(args[n], XtNplotLineStyle, AtLineSOLID), n++;
       line4 = XtCreateWidget("line4", atXYLinePlotWidgetClass, plotter, args, n);
       AtXYLinePlotAttachDoublePoints(line4, data, 1, nd);
      }
      else if(line5 == NULL) {
       n = 0;
       XtSetArg(args[n], XtNlegendName,    "test-data2"); n++;
       XtSetArg(args[n], XtNplotLineType,  AtPlotLINEPOINTS); n++;
       XtSetArg(args[n], XtNplotMarkType,  AtMarkTRIANGLE1); n++;
       XtSetArg(args[n], XtNplotLineStyle, AtLineDASHED), n++;
       line5 = XtCreateWidget("line5", atXYLinePlotWidgetClass, plotter, args, n);
       AtXYLinePlotAttachDoublePoints(line5, data, 1, nd);
      }
      else if(line6 == NULL) {
       n = 0;
       XtSetArg(args[n], XtNlegendName,    "test-data3"); n++;
       XtSetArg(args[n], XtNplotLineType,  AtPlotLINEIMPULSES); n++;
       XtSetArg(args[n], XtNplotMarkType,  AtMarkDIAMOND); n++;
       XtSetArg(args[n], XtNplotLineStyle, AtLineSOLID), n++;
       line6 = XtCreateWidget("line6", atXYLinePlotWidgetClass, plotter, args, n);
       AtXYLinePlotAttachDoublePoints(line6, data, 1, nd);
      }
      errmsg = NULL;
    }
    else {
      free(data);
      sprintf(msg, "Warning: Cannot not read %d items from data file %s", nd, filename);
      errmsg = msg;
    }
  }
  else {
    sprintf(msg, "Warning: Cannot open data file %s", filename);
    errmsg = msg;
  }
```

```
  return(errmsg);
}

/*
 *      The mainmenu callbacks
 */

extern int exit();

/* callback from dialog */
static char *plotfilename = NULL;

/* file_pulldown: quit_button */
void quit_callback()
{
     exit(0);
}

/* file_pulldown: read_button */
void cancelread_callback();

void readdata_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  XmSelectionBoxCallbackStruct *call_data;
{
  char *msg;

  XtRemoveCallback(dialog, XmNokCallback, readdata_callback, NULL);
  XtRemoveCallback(dialog, XmNcancelCallback, cancelread_callback, NULL);
  XmStringGetLtoR(call_data->value, XmSTRING_DEFAULT_CHARSET, &plotfilename);
  if(plotfilename) {
    if(msg = ReadPlotData(plotfilename))
      Popup_warning(msg);
    XtFree(plotfilename);
    plotfilename = NULL;
  }
}

void cancelread_callback()
{
  XtRemoveCallback(dialog, XmNokCallback, readdata_callback, NULL);
  XtRemoveCallback(dialog, XmNcancelCallback, cancelread_callback, NULL);
}

void read_callback()
{
  if(line4 && line5 && line6) {
    Popup_warning("Warning: Cannot create more than 3 lines");
  }
  else
    Popup_dialog(readdata_callback, cancelread_callback,
              "Plotter Demo Datafile Selection", "*.data");
}

/* file_pulldown: print_button */
static char *printfilename = NULL;

void cancelprint_callback();

void psprint_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  XmSelectionBoxCallbackStruct *call_data;
{
```

```
  char msg[256];

  XtRemoveCallback(dialog, XmNokCallback, psprint_callback, NULL);
  XtRemoveCallback(dialog, XmNcancelCallback, cancelprint_callback, NULL);
  XmStringGetLtoR(call_data->value, XmSTRING_DEFAULT_CHARSET, &printfilename);
  if(printfilename) {
    AtPlotterGeneratePostscript(printfilename, (AtPlotterWidget) plotter,
                              "Sample Plot", 50, 50, 480, 320, False);
    XtFree(printfilename);
    printfilename = NULL;
  }
}

void cancelprint_callback()
{
  XtRemoveCallback(dialog, XmNokCallback, psprint_callback, NULL);
  XtRemoveCallback(dialog, XmNcancelCallback, cancelprint_callback, NULL);
}

void print_callback()
{
  Popup_dialog(psprint_callback, cancelprint_callback,
              "Plot Demo PS Print", "*.ps");
}

/* plot_pulldown: askposition_button */
static Boolean is_asking = False;

void get_click_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  AtPointCallbackData *call_data;
{
  static char xstr[64], ystr[64];
  Arg args[2];

  sprintf(xstr, "%.4lf", call_data->x1);
  XtSetArg(args[0], XmNvalue, xstr);
  XtSetValues(x_text, args, 1);
  sprintf(ystr, "%.4lf", call_data->y1);
  XtSetArg(args[0], XmNvalue, ystr);
  XtSetValues(y_text, args, 1);
}

void askposition_callback()
{
  Arg args[4];

  if(is_asking) {
    XtRemoveAllCallbacks(plotter, XtNclickCallback);
    XtSetArg(args[0], XmNvalue, "");
    XtSetValues(x_text, args, 1);
    XtSetValues(y_text, args, 1);
    XtSetArg(args[0], XmNsensitive, False);
    XtSetValues(ask_label, args, 1);
    is_asking = False;
  }
  else {
    XtAddCallback(plotter, XtNclickCallback, get_click_callback, NULL);
    XtSetArg(args[0], XmNsensitive, True);
    XtSetValues(ask_label, args, 1);
    is_asking = True;
  }
}
```

```
/* plot_pulldown: getposition_button */
static Boolean is_positioning = False;

void get_motion_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  AtPointCallbackData *call_data;
{
  static char xstr[64], ystr[64];
  Arg args[2];

  sprintf(xstr, "%.4lf", call_data->x1);
  XtSetArg(args[0], XmNvalue, xstr);
  XtSetValues(x_text, args, 1);
  sprintf(ystr, "%.4lf", call_data->y1);
  XtSetArg(args[0], XmNvalue, ystr);
  XtSetValues(y_text, args, 1);
}

void getposition_callback()
{
  Arg args[4];

  if(is_positioning) {
    XtRemoveAllCallbacks(plotter, XtNmotionCallback);
    XtSetArg(args[0], XmNsensitive, False);
    XtSetValues(xpos_label, args, 1);
    XtSetValues(ypos_label, args, 1);
    XtSetArg(args[0], XmNvalue, "");
    XtSetValues(x_text, args, 1);
    XtSetValues(y_text, args, 1);
    is_positioning = False;
  }
  else {
    XtSetArg(args[0], XmNsensitive, True);
    XtSetValues(xpos_label, args, 1);
    XtSetValues(ypos_label, args, 1);
    XtAddCallback(plotter, XtNmotionCallback, get_motion_callback, NULL);
    is_positioning = True;
  }
}

void plotterform_callback(widget, client_data, event, cont)
  Widget widget;
  XtPointer client_data;
  XEvent *event;
  Boolean cont;
{
  Arg args[2];
  Boolean enter = (Boolean) client_data;

  if(enter) {
    XtSetArg(args[0], XmNshadowType, XmSHADOW_IN);
  }
  else {
    if(is_positioning && !is_asking) {
      XtSetArg(args[0], XmNvalue, "");
      XtSetValues(x_text, args, 1);
      XtSetValues(y_text, args, 1);
    }
    XtSetArg(args[0], XmNshadowType, XmSHADOW_OUT);
  }
  XtSetValues(plotter_frame, args, 1);
}
```

```
/* plot_pulldown: zoom_button */

static double oldxmin, oldxmax, minxlen;
static double oldymin, oldymax, minylen;

void get_drag_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  AtRectangleCallbackData *call_data;
{
  Arg args[4];
  char msg[256];
  static double x1, y1, x2, y2;
  double xl, yl;

  x1 = call_data->x11;
  x2 = call_data->x12;
  xl = fabs(x2 - x1);
  y1 = call_data->y11;
  y2 = call_data->y12;
  yl = fabs(y2 - y1);
  if(xl < minxlen) {
    sprintf(msg, "Warning: Zooming not possible for X axis (>%.0lf%%)", ZOOM);
    Popup_warning(msg);
  }
  else {
    XtSetArg(args[0], XtNmin, &x1);
    XtSetArg(args[1], XtNmax, &x2);
    XtSetArg(args[2], XtNautoScale, False);
    XtSetValues(xaxis, args, 3);
  }
  if(yl < minylen) {
    sprintf(msg, "Warning: Zooming not possible for Y axis (>%.0lf%%)", ZOOM);
    Popup_warning(msg);
  }
  else {
    XtSetArg(args[0], XtNmin, &y1);
    XtSetArg(args[1], XtNmax, &y2);
    XtSetArg(args[2], XtNautoScale, False);
    XtSetValues(yaxis, args, 3);
  }
}

void zoom_callback()
{
  Arg args[6];
  int n;
  static Boolean is_zooming = False;

  if(is_zooming) {
    XtRemoveAllCallbacks(plotter, XtNdragCallback);
    XtSetArg(args[0], XtNmin, &oldxmin);
    XtSetArg(args[1], XtNmax, &oldxmax);
    XtSetArg(args[2], XtNautoScale, True);
    XtSetValues(xaxis, args, 3);
    XtSetArg(args[0], XtNmin, &oldymin);
    XtSetArg(args[1], XtNmax, &oldymax);
    XtSetArg(args[2], XtNautoScale, True);
    XtSetValues(yaxis, args, 3);
    XtSetArg(args[0], XmNsensitive, False);
    XtSetValues(zoom_label, args, 1);
    is_zooming = False;
  }
  else {
    XtVaGetValues(xaxis, XtNmin, &oldxmin, XtNmax, &oldxmax, NULL);
```

```
    XtVaGetValues(yaxis, XtNmin, &oldymin, XtNmax, &oldymax, NULL);
    minxlen = (oldxmax - oldxmin) / ZOOM;
    minylen = (oldymax - oldymin) / ZOOM;
    XtAddCallback(plotter, XtNdragCallback, get_drag_callback, NULL);
    XtSetArg(args[0], XmNsensitive, True);
    XtSetValues(zoom_label, args, 1);
    is_zooming = True;
  }
}

/* plot_pulldown: legend_button */
void legend_callback()
{
  Arg args[2];
  Boolean l;

  XtVaGetValues(plotter, XtNshowLegend, &l, NULL);
  l = l ? False : True;
  XtSetArg(args[0], XtNshowLegend, l);
  XtSetValues(plotter, args, 1);
}

/* plot_pulldown: position_button */
void position_callback()
{
  Arg args[4];
  Boolean l;

  XtVaGetValues(plotter, XtNlegendLeft, &l, NULL);
  l = l ? False : True;
  XtSetArg(args[0], XtNshowLegend, True);
  XtSetArg(args[1], XtNlegendLeft, l);
  XtSetValues(plotter, args, 2);
}

/* plot_pulldown: pixmap_button */
void pixmap_callback()
{
  Arg args[2];
  Boolean l;

  XtVaGetValues(plotter, XtNusePixmap, &l, NULL);
  l = l ? False : True;
  XtSetArg(args[0], XtNusePixmap, l);
  XtSetValues(plotter, args, 1);
}

/* axis_pulldown: transform_button */
void transform_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  XmAnyCallbackStruct call_data;

{
  Arg args[2];

  if(AtAxisGetTransform(xaxis) == AtTransformLOGARITHMIC) {
    XtSetArg(args[0], XmNsensitive, True);
    XtSetValues(linearx_button, args, 1);
    XtSetArg(args[0], XmNsensitive, False);
    XtSetValues(logarithmicx_button, args, 1);
  }
  else {        /* LINEAR */
    XtSetArg(args[0], XmNsensitive, False);
    XtSetValues(linearx_button, args, 1);
```

```
    XtSetArg(args[0], XmNsensitive, True);
    XtSetValues(logarithmicx_button, args, 1);
  }
  if(AtAxisGetTransform(yaxis) == AtTransformLOGARITHMIC) {
    XtSetArg(args[0], XmNsensitive, True);
    XtSetValues(lineary_button, args, 1);
    XtSetArg(args[0], XmNsensitive, False);
    XtSetValues(logarithmicy_button, args, 1);
  }
  else {        /* LINEAR */
    XtSetArg(args[0], XmNsensitive, False);
    XtSetValues(lineary_button, args, 1);
    XtSetArg(args[0], XmNsensitive, True);
    XtSetValues(logarithmicy_button, args, 1);
  }
}

/* axis_pulldown/transform_pulldown: linearx_button */
void linearx_callback()
{
  Arg args[2];

  XtSetArg(args[0], XtNaxisTransform, AtTransformLINEAR);
  XtSetValues(xaxis, args, 1);
}

/* axis_pulldown/transform_pulldown: logarithmicx_button */
void logarithmicx_callback()
{
  Arg args[2];

  XtSetArg(args[0], XtNaxisTransform, AtTransformLOGARITHMIC);
  XtSetValues(xaxis, args, 1);
}
/* axis_pulldown/transform_pulldown: lineary_button */
void lineary_callback()
{
  Arg args[2];

  XtSetArg(args[0], XtNaxisTransform, AtTransformLINEAR);
  XtSetValues(yaxis, args, 1);
}

/* axis_pulldown/transform_pulldown: logarithmicy_button */
void logarithmicy_callback()
{
  Arg args[2];

  XtSetArg(args[0], XtNaxisTransform, AtTransformLOGARITHMIC);
  XtSetValues(yaxis, args, 1);
}

/* axis_pulldown/round button: roundx_button */
void roundx_callback()
{
  Arg args[2];
  Boolean r;

  XtVaGetValues(xaxis, XtNroundEndpoints, &r, NULL);
  XtSetArg(args[0], XtNroundEndpoints, r ? False : True);
  XtSetValues(xaxis, args, 1);
}

/* axis_pulldown/round button: roundy_button */
void roundy_callback()
```

```
{
  Arg args[2];
  Boolean r;

  XtVaGetValues(yaxis, XtNroundEndpoints, &r, NULL);
  XtSetArg(args[0], XtNroundEndpoints, r ? False : True);
  XtSetValues(yaxis, args, 1);
}

/* axis_pulldown: grid_button */
void grid_callback()
{
  Arg aargs[2];
  Arg gargs[2];
  Boolean gx, gy;

  XtVaGetValues(xaxis, XtNdrawGrid, &gx, NULL);
  XtVaGetValues(yaxis, XtNdrawGrid, &gy, NULL);
  if(gx || gy) {
    XtSetArg(aargs[0], XtNdrawGrid, False);
    XtSetArg(gargs[0], XmNsensitive, False);
  }
  else {
    XtSetArg(aargs[0], XtNdrawGrid, True);
    XtSetArg(gargs[0], XmNsensitive, True);
  }
  XtSetValues(xaxis, aargs, 1);
  XtSetValues(yaxis, aargs, 1);
  XtSetValues(subgrid_button, gargs, 1);
}

/* axis_pulldown: subgrid_button */
void subgrid_callback()
{
  Arg args[2];
  Boolean gx, gy, gn;

  XtVaGetValues(xaxis, XtNdrawSubgrid, &gx, NULL);
  XtVaGetValues(yaxis, XtNdrawSubgrid, &gy, NULL);
  gn = gx || gy ? False : True;
  XtSetArg(args[0], XtNdrawSubgrid, gn);
  XtSetValues(xaxis, args, 1);
  XtSetValues(yaxis, args, 1);
}

/* line_pulldown: linewidth_buttons 0..3 contained in client_data */
void width_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  XmAnyCallbackStruct call_data;
{
  Arg args[2];

  XtSetArg(args[0], XtNlineWidth, client_data);
  if(line1)
    XtSetValues(line1, args, 1);
  if(line2)
    XtSetValues(line2, args, 1);
  if(line3)
    XtSetValues(line3, args, 1);
  if(line4)
    XtSetValues(line4, args, 1);
  if(line5)
    XtSetValues(line5, args, 1);
  if(line6)
```

```
    XtSetValues(line6, args, 1);
}

/* line_pulldown: linetype_buttons. Type is contained in client_data */
void type_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  XmAnyCallbackStruct call_data;
{
  Arg args[2];
  AtPlotLineType type;

  switch((int) client_data) {
    case 0: type = AtPlotLINES;        break;
    case 1: type = AtPlotPOINTS;       break;
    case 2: type = AtPlotIMPULSES;     break;
    case 3: type = AtPlotSTEPS;        break;
    case 4: type = AtPlotLINEPOINTS;   break;
    case 5: type = AtPlotLINEIMPULSES; break;
    case 6: type = AtPlotBARS;         break;
    default: return;
  }
  XtSetArg(args[0], XtNplotLineType, type);
  if(line1)
    XtSetValues(line1, args, 1);
  if(line2)
    XtSetValues(line2, args, 1);
  if(line3)
    XtSetValues(line3, args, 1);
  if(line4)
    XtSetValues(line4, args, 1);
  if(line5)
    XtSetValues(line5, args, 1);
  if(line6)
    XtSetValues(line6, args, 1);
}

/* line_pulldown: linestyle_buttons. Style is contained in client_data */
void style_callback(widget, client_data, call_data)
  Widget widget;
  XtPointer client_data;
  XmAnyCallbackStruct call_data;
{
  Arg args[2];
  AtPlotLineStyle style;

  switch((int) client_data) {
    case 0: style = AtLineSOLID;         break;
    case 1: style = AtLineDOTTED;        break;
    case 2: style = AtLineDASHED;        break;
    case 3: style = AtLineDOTDASHED;     break;
    case 4: style = AtLineDOTTED3;       break;
    case 5: style = AtLineDASHED4;       break;
    case 6: style = AtLineDOTDASHED2;    break;
    default: return;
  }
  XtSetArg(args[0], XtNplotLineStyle, style);
  if(line1)
    XtSetValues(line1, args, 1);
  if(line2)
    XtSetValues(line2, args, 1);
  if(line3)
    XtSetValues(line3, args, 1);
  if(line4)
    XtSetValues(line4, args, 1);
```

```
  if(line5)
    XtSetValues(line5, args, 1);
  if(line6)
    XtSetValues(line6, args, 1);
}

/* help_pulldown: version_button */
void version_callback()
{
  Popup_help(version);
}

/* help_pulldown: copyright_button */
void copyright_callback()
{
  Popup_help(copyright);
}

/* help_pulldown: author_button */
void author_callback()
{
  Popup_help(author);
}

/*
 *      Handler for Xt warnings
 */

static void warning_handler(msg)
  char *msg;
{
  char buf[256];

  sprintf(buf, "Warning: %s", msg);
  Popup_warning(buf);
}

/*
 *      Build plotter and create/attach data
 */

void BuildPlotter()
{
  Arg args[2];
  XmString xms[1];
  Display *dpy;
  Window win;
  Pixmap ipm;

  /* Set the icon bitmap */
  dpy = XtDisplay(appshell);
  win = DefaultRootWindow(dpy);
  ipm = XCreateBitmapFromData(dpy, win, mplot_bits, mplot_width, mplot_height);
  XtSetArg(args[0], XmNiconPixmap, ipm);
  XtSetValues(appshell, args, 1);

  /* Set the title */
  xms[0] = (XmString) XmStringCreateLtoR(TITLE, XmSTRING_DEFAULT_CHARSET);
  XtSetArg(args[0], XmNlabelString, xms[0]);
  XtSetValues(title_label, args, 1);
  XmStringFree(xms[0]);

  /* Make plotter/axes/lines and data */
  MakePlotter(plotter_frame);
```

```
  /* Create the demo lines and attach data */
  if(display_demo_lines == True) {
    MakeDemoLines();
    MakeData();
    AttachData();
  }

  /* Set a warning message handler */
  XtAppSetWarningHandler(application_context, warning_handler);
}
```

4.11. The Test Program sample_prog

Gregory Bond's test program `sample_prog.c` from version 5 of the plotter widget set should be included in the directory `athena`.

```
/*
 * A simple sample test program
 *
 * Copyright 1991 by Burdett, Buckeridge & Young Ltd.
 */

/* First, the various headers required */

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Form.h>

#ifdef _AtDevelopment_
#include "Plotter.h"
#include "LinePlot.h"
#include "BarPlot.h"
#include "Axis.h"
#else
#include <At/Plotter.h>
#include <At/LinePlot.h>
#include <At/BarPlot.h>
#include <At/Axis.h>
#endif

/*
 * The application data - weekly sales data for Widgets.
 */
struct profits {
     /* From database */
     long volume;
     float buy_price;
     float sell_price;
     /* calculated values */
     double sales;
     double cost;
     double profit;
};

#define NUM_WEEKS 52

struct profits profits[NUM_WEEKS];

/*
 * Read from the database - a dummy routine, we'll use random numbers
 */
void ReadFromDatabase()
{
     int i;
     extern int rand();

     for (i = 0; i < NUM_WEEKS; i++) {
         /* Volume between 1000 and 5095 */
         profits[i].volume = (rand() & 0xfff) + 1000;
         /* Sell price between $15 and $25 */
         profits[i].sell_price = (double)(rand() & 0xffffff) /
              0xffffff * 10.0 + 15.0;
         /* Buy price between $5 and $15 */
         profits[i].buy_price = (double)(rand() & 0xffffff) /
```

```
              0xffffff * 10.0 + 5.0;
     }
}

/*
 * Calculate the profit from the above data
 */

void CalculateProfit()
{
     int i;

     for (i = 0; i < NUM_WEEKS; i++) {
         profits[i].sales = profits[i].volume *
              profits[i].sell_price;
         profits[i].cost = profits[i].volume * profits[i].buy_price;
         profits[i].profit = profits[i].sales - profits[i].cost;
     }
}


/*****************************************************************
 *
 * This is the interesting bit.  Make a simple graph, profits on left,
 * volume on right, weeknumber on bottom, profit a line graph, volume
 * a bar graph.
 */

Widget plotter, xaxis, yaxis, y2axis, line, bar;

void MakePlotter(parent)
Widget parent;
{
     double thousandth = 1.0e-3;
     plotter = XtVaCreateManagedWidget("plotter", atPlotterWidgetClass, parent,
                                  XtNtitle, "Widget Sales & Profits",
                                  XtNwidth, 600, XtNheight, 450,
                                  NULL);
     /* Note axis objects are unmanaged */
     xaxis = XtVaCreateWidget("xaxis", atAxisWidgetClass, plotter,
                          XtNlabel, "Week Number",
                          XtNticFormat, "%.0f",
                          NULL);
     /* Note Y axis is vertical */
     yaxis = XtVaCreateWidget("yaxis", atAxisWidgetClass, plotter,
                          XtNvertical, True,
                          XtNlabel, "Profit ($1,000's)",
                          XtNticMultiplier, &thousandth,
                          NULL);
     /* Note Y2 axis is vertical and mirrored */
     y2axis = XtVaCreateWidget("y2axis", atAxisWidgetClass, plotter,
                           XtNvertical, True,
                           XtNmirror, True,
                           XtNlabel, "Volume Shipped (1,000s)",
                           XtNdrawGrid, False, /* Two grids is confusing */
                           XtNticMultiplier, &thousandth,
                           XtNticFormat, "%4.1f",
                           NULL);
     /* Now attach axes to the parent */
     XtVaSetValues(plotter, XtNxAxis, xaxis,
                XtNyAxis, yaxis,
                XtNy2Axis, y2axis,
                NULL);

     /* Now create the plots */
```

```
     /* The volume is scaled against y2 axis */
     bar = XtVaCreateWidget("vol", atBarPlotWidgetClass, plotter,
                       XtNlegendName, "Volume",
                       XtNuseY2Axis, True,
                       XtNshading, AtGRAY3, /* For better PS */
                       NULL);
     /* But profit against Y axis */
     line = XtVaCreateWidget("profit", atLinePlotWidgetClass, plotter,
                        XtNlegendName, "Profit",
                        XtNlineWidth, 2,
                        NULL);
}

/*****************************************************************
 *
 * Attach the application data to the plot children
 */

void AttachData()
{
     AtLinePlotAttachData(line, (XtPointer)&profits[0].profit,
                      AtDouble, sizeof (struct profits),
                      1, NUM_WEEKS);
     AtBarPlotAttachData(bar, (XtPointer)&profits[0].volume,
                     AtInt, sizeof (struct profits),
                     1, NUM_WEEKS);
}

/*****************************************************************
 *
 * This is the boring bit: a form with two buttons and the plotter as
 * children
 */

Widget form, quitbutton, printbutton;

extern int exit();

void quit_callback()
{
     exit(0);
}

void print_callback()
{
     AtPlotterGeneratePostscript("ps.out", (AtPlotterWidget)plotter,
                            "Widget Profits", 0, 0, 400, 250, False);
     fprintf(stderr, "Plot dumped in ps.out0);
}

void MakeForm(parent)
Widget parent;
{
     form = XtVaCreateManagedWidget("form", formWidgetClass, parent, NULL);
     quitbutton = XtVaCreateManagedWidget("quit", commandWidgetClass, form,
                                  XtNlabel, "Quit",
                                  XtNleft, XtChainLeft,
                                  NULL);
     XtAddCallback(quitbutton, XtNcallback, quit_callback, NULL);
     printbutton = XtVaCreateManagedWidget("print", commandWidgetClass, form,
                                   XtNlabel, "Print",
                                   XtNfromHoriz, quitbutton,
                                   NULL);
     XtAddCallback(printbutton, XtNcallback, print_callback, NULL);
}
```

```
int main(ac, av)
Cardinal ac;
char **av;
{
     Widget appShell;
     XtAppContext app;

     appShell = XtAppInitialize(&app, "Test", NULL, 0, &ac, av,
                          NULL, NULL, 0);

     MakeForm(appShell);
     MakePlotter(form);
     /* Make the plotter below the buttons! */
     XtVaSetValues(plotter, XtNleft, XtChainLeft,
                 XtNfromVert, quitbutton,
                 NULL);

     /* Now read the data and calculate it */
     ReadFromDatabase();
     CalculateProfit();

     /* Now attach the data to the plot widget */
     AttachData();

     XtRealizeWidget(appShell);

     XtAppMainLoop(app);
     /*NOTREACHED*/
     return 0;
}
```