Use of the Matplotlib Driver Code

9 November 2020

The code described here is intended to allow one to make plots in Python/matplotlib using a user interface rather than needing to write code each time one needs to make a plot. The specific target usage is to be able to read data values from an ascii file and plot these, although one can also make data values within the Python interpreter and plot those values with the code herein.

The code depends on astropy, matplotlib, numpy, scipy, and tkinter. The tkinter functions come standard in Python, while one needs to install astropy and matplotlib for the use with this code. Installation of matplotlib will also install numpy and scipy. The installation of matplotlib is described at https://matplotlib.org/users/installing.html#installing-an-official-release . The astropy package is only used for reading/writing images in the FITS format that is commonly used in astronomy. This is only used in the code for two-dimensional histograms of plots, but the capability is present in case it is needed. The installation of the astropy package is described at https://docs.astropy.org/en/stable/install.html .

This driver code is loosely based on the long-standing xmgrace code that the author has used in various Unix-type systems for decades. While the xmgrace code is superior to this code, it depends on X11/Motif and it appears that these packages are not going to be viable much longer.

The code, matplotlib_user_interface.py, was originally intended to be run from the command line. When packaged for github this becomes the secondary mode for use of the code. When the code is run from the command line the main window appears. One then can read in data values or make data values for plotting. Various menus give the user control over quite a few aspects of the plotting. A similar result can be obtained from the python interpreter with a couple of lines.

Warning for MacOS 10.14/Python 3.7 Users

The code here uses normal Tkinter calls and the TkAgg “backend” in matplotlib. This is a standard way to make plots within a window. There is an issue in the MacOS 10.14 operating system that crashes the user session when these Tkinter functions are called under Python 3.7. This issue is not present in earlier versions of Python or in Python 3.8. If one is using MacOS 10.14 “Mojave” one needs to avoid trying to use the code with Python 3.7. Either downgrading to Python 3.6 or upgrading to Python 3.8 fixes this issue. There does not appear to be a work-around within Python 3.7 for this version of MacOS. The author has not seen this type of issue with previous versions of MacOS, or on Linux systems, for these tkinter programs. The problem seems specific to Python 3.7 in this operating system.

Installation of the Code

The code needs to be installed via the usual setup.py utility. One needs to download the code into a directory and then issue the command

```
python setup.py install
```

which will then install the code into one's python distribution. This allows the code to be imported and run in the python interpreter. This is done in the upper level directory, not the in `src` sub-directory where the code files are stored.

If one also wishes to use the program from the command line, one then has to arrange that the directory where the code files are stored, the `src` sub-directory in the package files, is part of the `$PATH` and `$PYTHONPATH` environment variables. This can be done on a Linux or MacOS system using the `setenv` or `export` variables depending on whether one is using the tcsh/csh shell or is using the bash shell. The format of the command would be

setenv PATH /your/path/to/the/files:${PATH}

or

export PATH='/your/path/to/the/files:$PATH'

for tsch/csh and bash respectively. This assumes that the $PATH variable is already defined. In the above "/your/path/to/the/files" should be replaced by the actual path to where the code files are stored on your system.

Running the Interface From Within Python

The code can be called from within Python. This will bring up the interface for use. One can load data sets into the plot at will from the Python interpreter. Instructions for how to do this are given in the docstring at the start of the code, and are also given in this section. The plot interface is a Python object. Almost all of the functions use the one object as the only parameter in the case. The object has a large number of variables stored within it and these are used in the different routines where calculations are done or the data values are manipulated.

The simplest use of the code from within Python would be as follows, where a simple pair of numpy vectors named `xvalues` and `yvalues` are made for the plot:

```
>>>> import matplotlib_user_interface as mui
>>>> root, myplot = mui.startup()
>>>> xvalues = numpy.arange(0, 10)
>>>> yvalues = xvalues*xvalues
>>>> myplot.add_set(xvalues, yvalues)
```

(repeat for multiple sets, if needed)

```
>>>> mui.make_plot.make_plot(myplot)
```

This last statement directs the code to update the plot display in object myplot with the data values that have been defined. One can add as many sets of data values to the plot as needed by repeating the add_set calls. Each one adds the data values into the plot object. One can also manipulate the plot properties via different types of either calls or variable assignments within the object. However, it is generally better to use the interface for these activities. The `startup` utility routine is provided to simplify setting up the interface from Python.

If one wants to bring up the interface from within python without any data sets being defined first, so that one can read in data values from files in the same way as when running the code from the command line, one only needs the first two lines.

```
>>>> import matplotlib_user_interface as mui
>>>> root, myplot = mui.startup()
```

After the second of these two commands the plot window will appear and be ready for use.

If one wishes to do all the steps manually the following somewhat longer set of commands can be used:

(1) import the required packages tkinter and matplotlib_gui_code

```
>>> import tkinter as Tk
>>> import matplotlib_user_interface
```

The next step defines a window in tkinter to use for the plotting:

```
>>> root = Tk.Tk()
```

This produces a window with nothing in it.

(2) Generate x and y data as numpy arrays as needed

Assume that "xvalues" and "yvalues" hold the data values for a plot. These need to be numpy arrays and should not be lists. One simple example would be

```
>>>> xvalues = numpy.arange(1, 101, 1)
>>>> yvalues = numpy.log10(xvalues * xvalues)
```

(3) Define the plotting object

```
>>> myplot = matplotlib_user_interface.PlotGUI(root)
```

This fills in the buttons/functions in the window. The functionality is then available.

(4) Add a set or sets to the plot. Any number of sets up a limit (currently 100) can be added. See the `add_set` subroutine doc string for the possible use of error bars in the set. The value `self.max_sets`, which would be `myplot.max_sets` here, determines the maximum number of sets that can be handled.

```
>>> myplot.add_set(xvalues, yvalues)
```

(5) Tell the code to plot or replot. This call can be used as many times as required.

```
>>> mui.make_plot.make_plot(myplot)
```

The `add_set` routine takes as input the x and y data values and makes a new set. Hence one can mix reading in values and generating values inside python as needed by following the above set of steps. There are additional parameter values one can pass to this routine, for error values in the data points.

Running Within a Script

When running in a script rather than at the python interpreter, one needs to add

```
root.mainloop()
```

after making the call to mui.make_plot.`make_plot(myplot)` with the sets, to keep the window in place. Otherwise the window may disappear after the `make_plot` call if nothing else is being done in the script.

Running From the Command Line

Use of the code from the command line was the original default for the interface, since that is also how the "xmgrace" code operated. Although the code now can be installed and used directly within Python as described in the previous section, the option of using it from the command line is still available.

In a Linux/Unix/MacOSX type of system the code can be started from the command line. The code is a set of programs; the main program, named matplotlib_user_interface.py, can be run by either a command such as

```
prompt> python matplotlib_user_interface.py
```

or invoking the code directly if the file is executable

```
prompt> matplotlib_user_interface.py
```

In the above "prompt>" represents the command prompt in a terminal widow or the like. The command only works if the program files are available in the directories included in the `$PATH` variable.

Use of the User Interface

When the code is started, either via the command line or within the Python interpreter, the main window comes up. Figure 1 shows the appearance of the window on a MacOS system. The salient points are that there is the main plot area at right, controls at left, and action meus where most of the work is done.

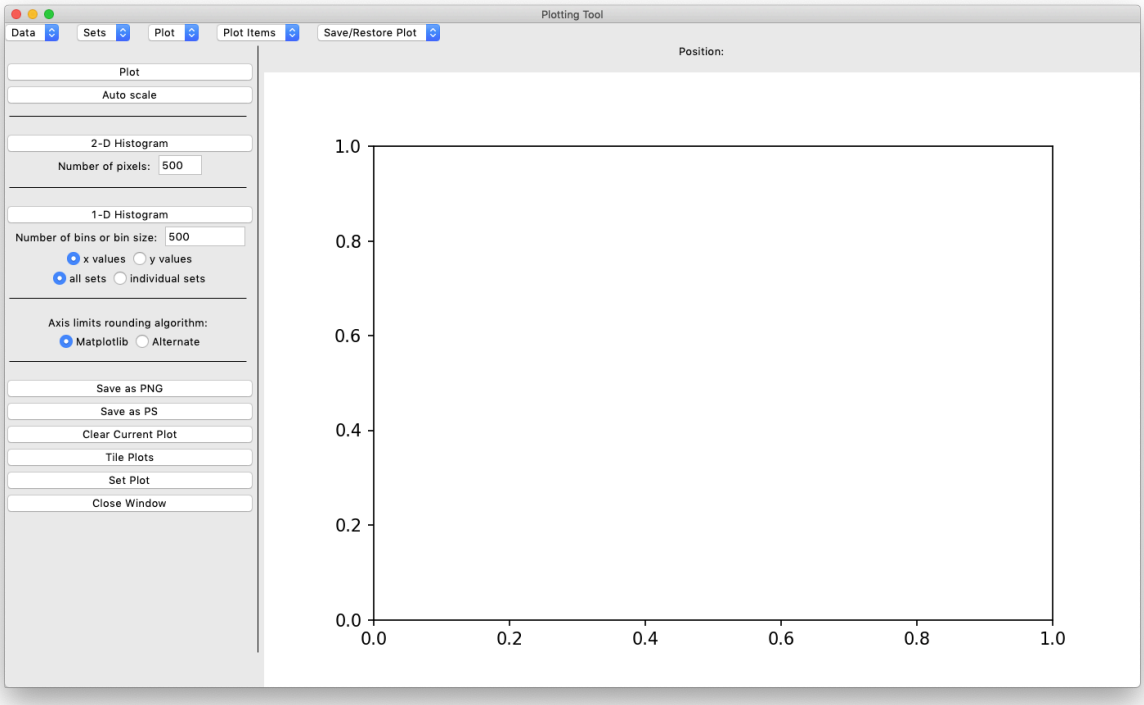

Figure 1: The initial appearance of the main window under a MacOS system.

Reading in Data Values

The first action to carry out is the creation of a "data set" for the plot. This can be done in several ways, using the "Data" menu at upper left. The Data menu options are

a) Read Data: read in data values from some type of formatted ascii file
b) Create Values by Formula: define a data set using a formula
c) Create Values in Widget: bring up a text window in which data values can be entered
d) Write Data: write the current data values to an ascii file

The normal situation is assumed to be reading in data from an ascii file. The data needs to be in columns separated by spaces; one can have comment lines with any the characters '#', '!', or '\' at the start of the line. Currently the code does not allow the user to define the character that is used to separate columns. The reading of data is done via the `numpy.loadtxt` function. The code brings up a window as shown in Figure 2. One can select a file using the normal file selection window by clicking on the "Select File" button at lower left. The string in the "Data File Filter" field at the top of the page is used to select the files of interest. (Note that this does not always seem to work well in the standard Tkinter FileSelection dialog widget.). Once a file is selected the code parses the first non-comment line to determine how many columns are

readable in the file. One then can select columns for the x and y data values, plus one or two uncertainty columns for x and for y as needed. The "Set Type" menu gives a selection of possible input schemes.

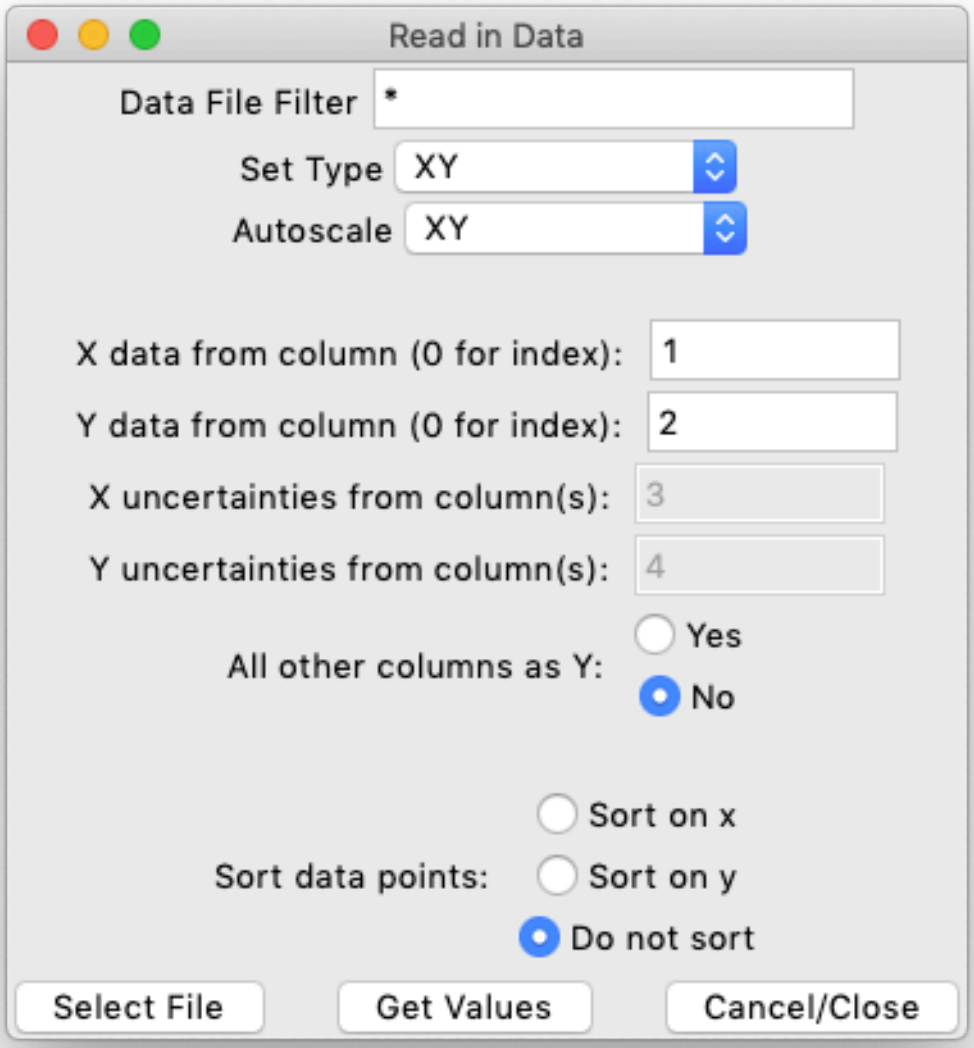

Figure 2: The window for reading in data from an ascii file.

Note that the column numbers start from 1 rather than from 0 as in Python. A value of 0 in the column entry box causes the code to use the index of the column values as the x or y variable. Once a file has been selected and the number of available columns has been determined, the window appearance is changed to let the user know how many columns are nominally available, as in Figure 3. Note that the parsing of the initial number of columns does

not guarantee that the columns can be read in successfully. If there are irregularities of the column structure within the file the reading of values may fail.

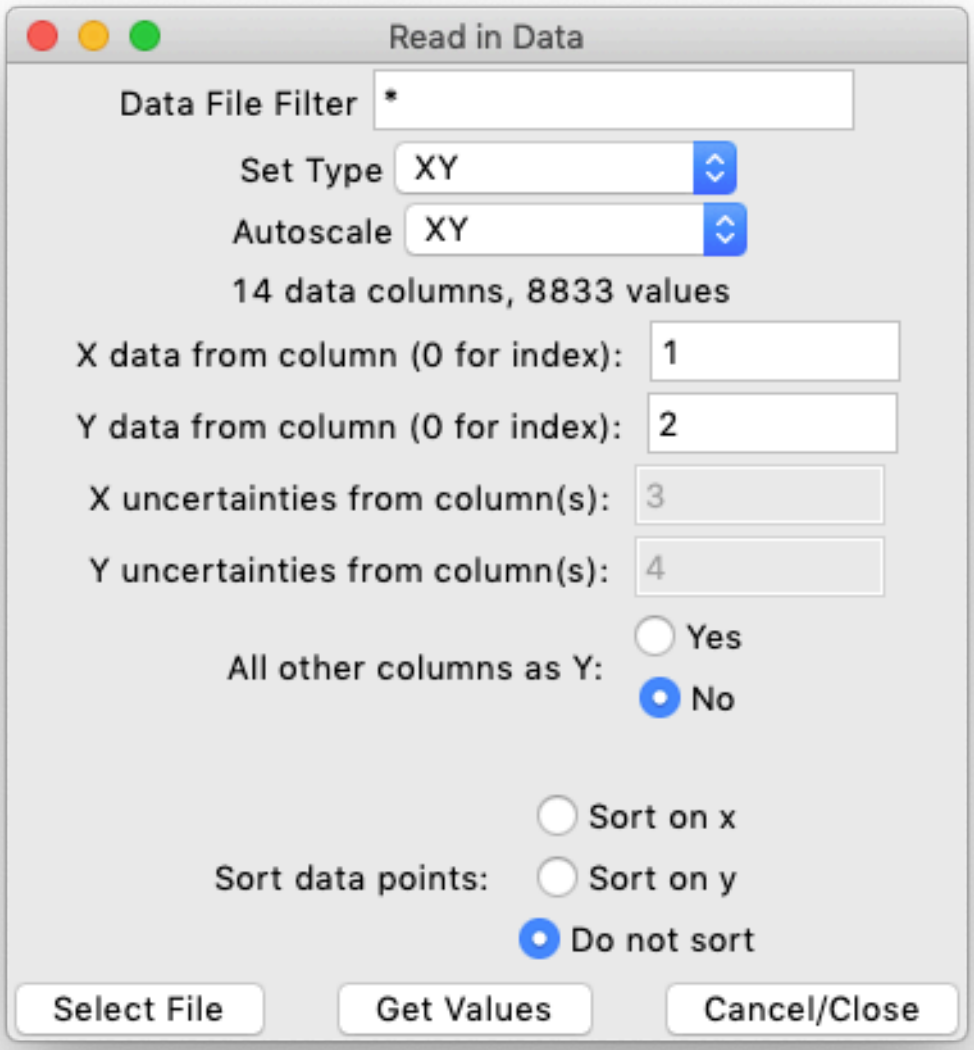

Figure 3: The appearance of the data read-in window once a file has been selected and the number of available columns have been determined.

One can set the "Set Type" using the first of the pull-down menus. One can select from the set of possibilities 'XY', 'XdXY', 'XYdY', 'XdXYdY', 'XYdYdY', 'XdXdXY', and 'XdXdXYdYdY'. In these items 'dX' marks an uncertainty in the x values, and 'dY' marks an uncertainty in the y values. One can have a single uncertainty that applies both positive and negative from the data value, or one can have two uncertainty values in which case the first one

is the uncertainty going in the negative direction and the second one is the uncertainty going in the positive direction. One need to provide a suitable number of columns to read that matches the menu entry. Hence if one selects 'XYdY' then one needs to specify one column for x, one for y, and one for the uncertainty in y, Δy. Once one has set the columns to use one can read the data in with the "Get Values" button. If the data cannot be read in an error pop-up message is displayed. If it can be read, the values are plotted and the appearance of the main plot area changes in response.

One can read in several sets of data from a single input file or read from a series of files one by one. One can also select whether to autoscale the plot when each data set is read in, using the "Autoscale" menu. The menu choices are to scale in both x and y, the default, to scale only in x, to scale only in y, or to not autoscale the plot. In the latter case the current range of the plot is not affected by the new data that is read in, and one may not see the data points in the plot.

Creating Values by a Formula

One can also define a formula for the data values that is evaluated within the code. This option produces a set of x and y values. One defines a numpy `arange` vector of values to start with, and then one can use these values in a calculation using normal math or numpy operations to make the data x and y values. Figure 4 shows the way the window looks when it first appears.

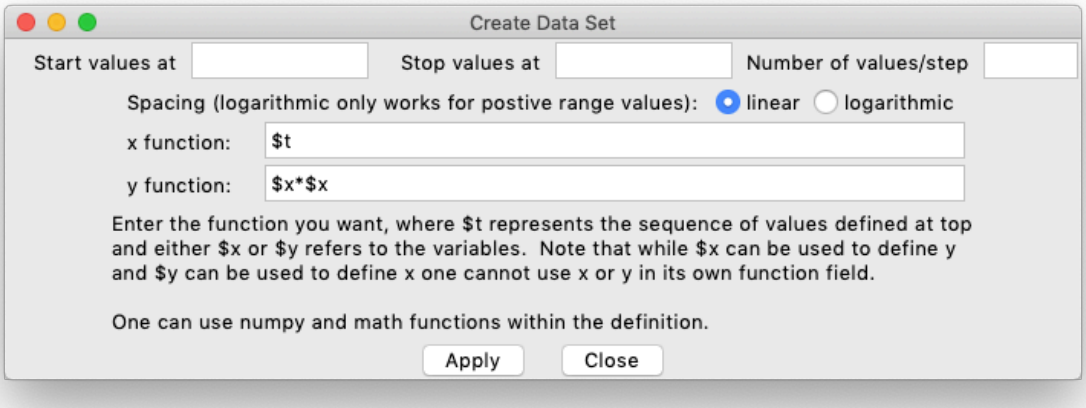

Figure 4: The window for creating a data set via a formula.

The idea is to first set the sequence of values by entering the start and stop values plus either a step value or the number of points in the upper three entry fields. The default is linear steps in the sequence. If the value in the third entry area at the top is an integer it is taken to be the number of values, and if it is a floating-point value it is taken to be the step size. One can make the sequence logarithmic, with a constant ratio of values from one point to the next, with the radio button option. The sequence is symbolized as $t in the function string fields. One can then use normal operations in the function fields to make the values.

The code allows the user to use normal arithmetic operations, any numpy operation if expressed in a form such as `numpy.sqrt($t)` within the entry field, and the normal math operations. Any other operations are not allowed to safeguard the code from arbitrary

commands being executed. Once the function is defined, the "Apply" button attempts to parse the fields and generate the data set. If it fails a pop-up window is displayed to alert the user to an issue.

Creating Values in a Window

The third menu option 'Create Values in Widget' brings up a Tkinter scrolled text window. Within the window one can type in rows of x and y values with spaces separating the columns. Only two columns are allowed, the first value is assumed to be an x value in the set and the second one is assumed to be a y value in the set. This is of limited utility compared to creating a file with data values outside the code and reading the values in, but is useful for some types of quick data entry.

Working With Plots

Once a set is read in the plot area is regenerated with the new data points plotted. One might see a plot such as in Figure 5. There is in this instance one data set with 2783 data values as read in from a particular file. The plot is made in the simplest way using the matplotlib defaults for most parameters. The exception is that symbol which is taken from a symbol values and colours so that one can read in many sets without any conflict in the symbols that are used. If one had read in uncertainty values for the plot then the points would have error bars in x or y or both.

Normally the code uses the matplotlib automatic determination of the x and y ranges for the data when a plot is first created. While in many instances this is fine, in some cases one may wish for an alternative rounding of values. The radio button selecting the rounding method can set the rounding to use the matplotlib algorithm, which is the default, or an alternative method. The alternative rounding rounds off the x and y limits to one significant digit. That is more likely to be useful if one uses logarithmic scaling in the x or y axes of the plot.

Once some sets are in place one can use the "Sets" and "Plot menus to control aspects of the plot. The main menu items for control of the plotting are the "Sets/Set Properties" and "Plot/Plot Parameters" entries. These are respectively to control the set display options and to control the plot and axes parameters.

Set Properties

The set properties, in particular the set symbol, line, and legend values used, are controlled in this window. The appearance of the window is shown in Figure 6. The menu at top allows one to select which set to work with. Below that are the set parameters. One has menus to select the symbol type and the line type. One can select a colour for the line and the symbol. One can set the symbol size and the line width in the entry fields. There are control buttons for whether the set is displayed in the plot or not, for whether error bars are plotted or not, and for whether the set is listed in the plot legend.

The code currently does not allow the symbol and line colour to differ for points within a given individual set. One would have to read in the same set twice and display it once with a symbol and once with a line to get different colours in the two cases. Similarly, to get error bars

that are a different colour than the symbol and line is not possible in a single set in the code currently.

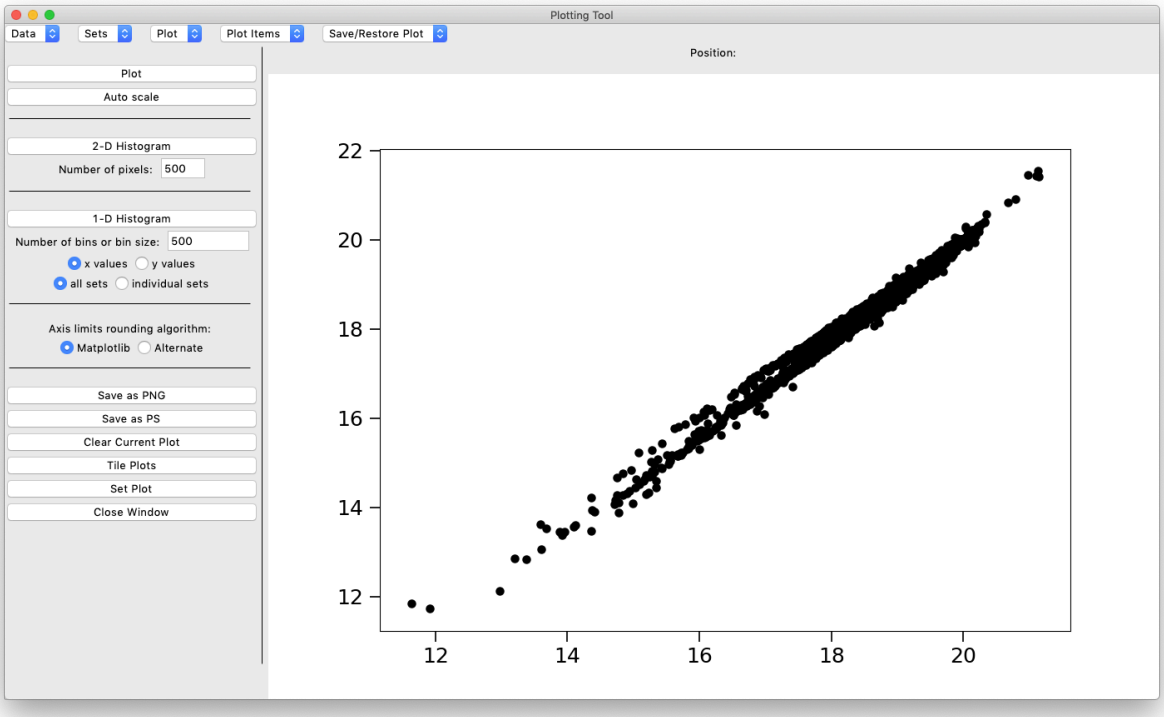

Figure 5: The plot window after a set of values has been read in. In this case there is one set of values in the plot. It has the default symbol and colour. If multiple sets are read in, they each will have a distinct symbol/colour combination.

The Label entry field is the text that is to be shown in the plot legend if that is displayed. The default text lists the columns used and the file name from which the values were read. This is not very useful in general so one would usually change the legend label before activating the legend in the plot.

Any changes to the values in the fields needs to be applied with the "Apply" button before either closing the window or switching to another set in the menu at the top. If one does not apply the values they are lost when either of these actions is carried out. Any changes that are applied will be seen immediately in the main plot window.

The full set of matplotlib symbols is available in the 'Symbol' menu, along with None to have no symbol displayed. Similarly the four normal matplotlib line types are available in the 'Line' menu along with an option of None to have no line. For the colours a set of 10 named colours are given in the menu along with the "Select" option. When "Select" is chosen the standard tkinter colour chooser window comes up to allow the user to select an arbitrary colour for the data set.

The Plot Control Window

Figure 7 shows the appearance of the main plot control window. This is where the axis properties and a few of the general plot properties are defined or changed. At right are a large

number of radio button controls for aspects of the axes. One can for example change the x or y axis from a linear axis to a logarithmic one with the radio buttons. One can select for tick marks on both axes or the direction of the tick marks. One can have a separate x axis or y axis opposite the main one. One can hide ticks or axis labels. All these can be selected with the radio buttons but will not have any effect until the 'Apply' button is activated.

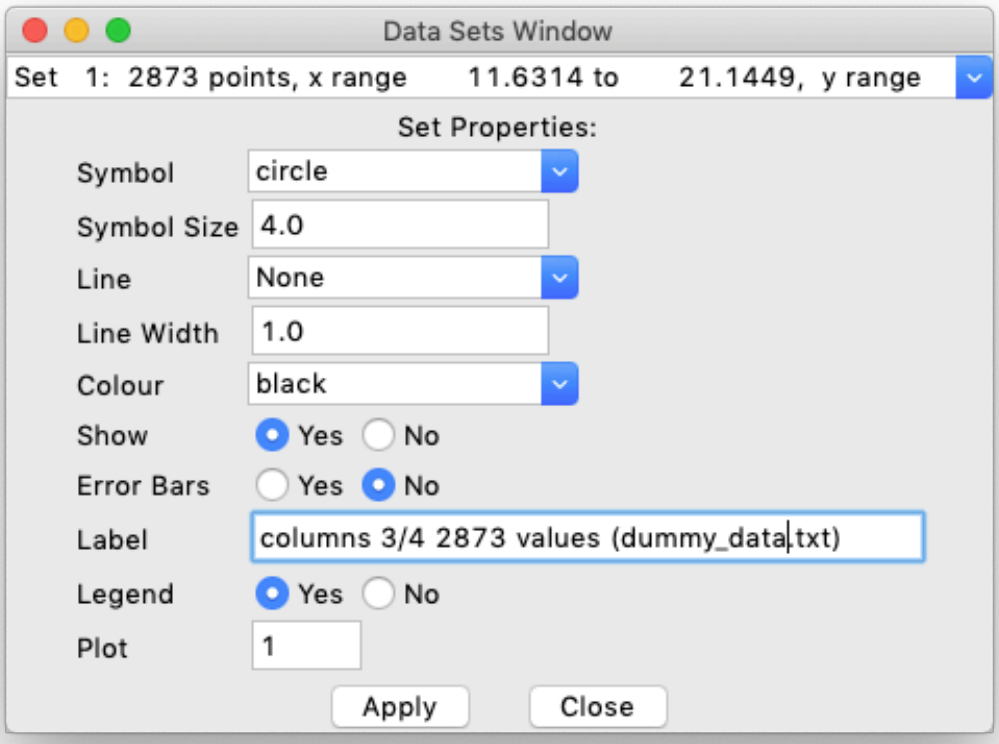

Figure 6: The set properties control window.

One unusual option is the 'hybrid log' axis type. Such an axis is linear between –10 and +10 and logarithmically scaled outside this range. Such a case is useful in instances where there is a large data range but where one is concerned about zero values. A normal logarithmic plot will not handle either zero values or negative values properly. This type of plot is able to handle both of these possibilities. This is a not regular practice, but is useful particularly for plotting histograms where one has to look at both large values and where the values are zero in the same plot. Note that the hybrid log option overrides the logarithmic option which overrides the linear option. One should in general have only one of these on at a time.

In the left part of the widget one is able to set the axis limits, axis labels, the title, and control some of the tick mark properties. This is also where the plot legend can be turned on or off. One can set a thickness for the plot frame. That is one option that generally improves the appearance of the plot over the default of having no frame.

The window also has fields wherein one can set the minor tick intervals in the x and y axes. The Matplotlib default to have no minor ticks is often not what one wants in a plot.

In the title and axis label fields one can get special symbols in the usual way for matplotlib, using TeX syntax in the string. See the matplotlib documentation section "Writing Mathematical Expressions" https://matplotlib.org/3.3.0/tutorials/text/mathtext.html for more details about how this can be done. The string in the entry field of the window is applied as written so one can use the TeX functions in the normal manner.

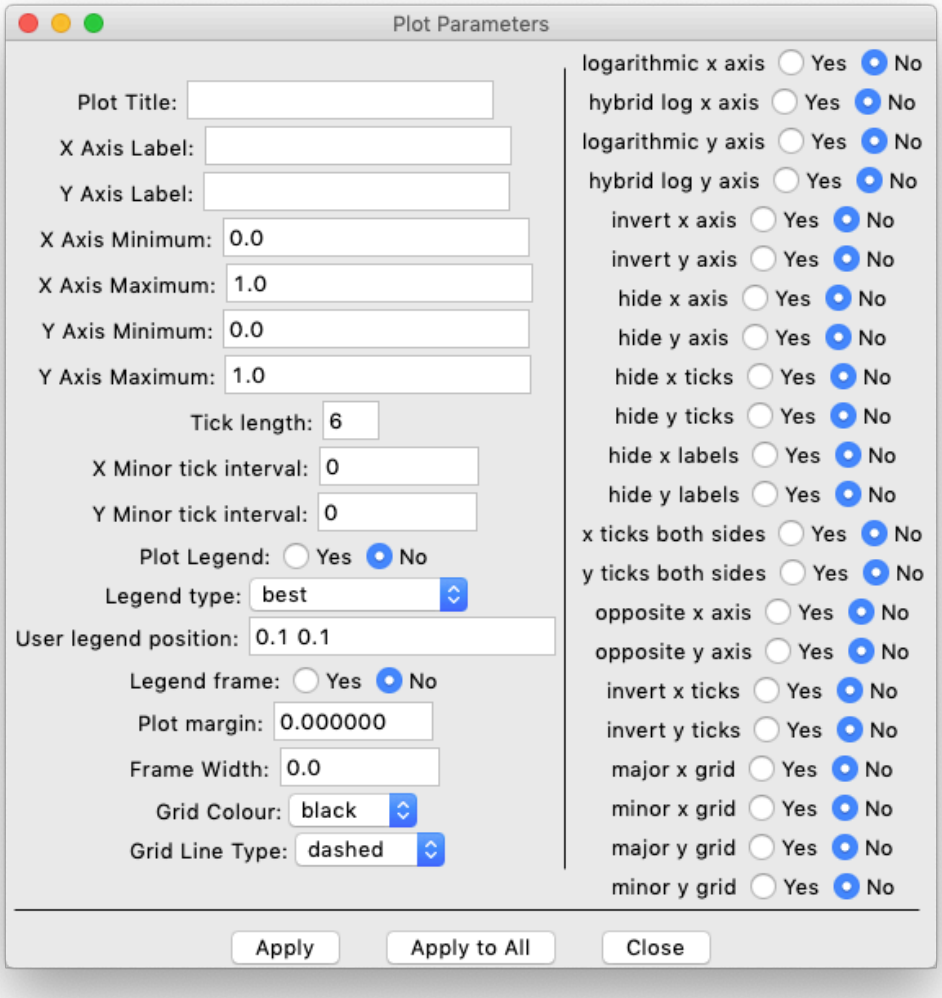

Figure 7: The main plot control window. This is used to deal with axes and plot properties in general.

The 'Plot margin' entry field allows one at direct matplotlib to leave more margin area around the current plot. This may be used along with the font size option if the axis labels are long and one has trouble fitting the plot properly into the plot window. The font size can be set along with the other font parameters with the 'Set Font' item in the 'Plot' menu. The code currently has only one font that is applied to all labels. One cannot for example vary the x and y axis font sizes independently in this code.

The window as an ‘Apply to All’ button as well as the ‘Apply’ button. If one has multiple plots this button can be used to apply a set of parameters to all the plots. This is a bit dangerous in general if one has many plots, but is convenient in some circumstances.

Use of Multiple Plots

The previous discussion has alluded to the possibility of having more than one plot active at a given time. The normal circumstance is to have a single plot and hence only one set of plot parameters. However, one may need to tile a few plots into a single display. This is done with the ‘Tile Plots’ item in the ‘Plot’ menu. If selected a small window comes up allowing the user to enter the number of plots in the x direction and the number of plots in the y direction. Only one plot is active at a given time, and one can set this number in the plot tile window or with commands in the Plot menu. Note that numbers are indexed to 1 here, unlike the normal Python internal counting.

If the plots are tiled one may want to add space between the plots to make them look better and avoid overlap. One is likely to also want to reduce the font size so the axis labels and tick labels do not overlap adjacent plots.

Each data set is associated with a single plot. One can move a data set between plots by changing the plot number in the ‘Set Properies’ window at the bottom. The code does not have any general mechanism for duplicating data sets between plots, which is one of the properties that the xmgrace code has.

One can use any key input to select the plot where the cursor is currently situated if there are multiple active plots. Or one can manually enter the plot number when the ‘Set plot’ item in the ‘Plot’ menu is selected.

There is an option to hide a plot. For example, if one wants three plots in a figure one would need to make a 2 by 2 grid of plots and then hide one of them. If a plot is hidden the parameters are preserved but the plot is not drawn. One can toggle a plot on and off at will.

Drawing Objects

The code allows one to define labels, lines, vectors, ellipses, and boxes on the plot. The general handling of all of these is similar. One has menu items to create any of these items. When activated, the code waits for button press events to signal that the item needs to be created. For a label only one button press event is needed, to mark the label position. For the other items two button press events are needed to define the two extremes of the object. This would be the initial and final points for lines and vectors or the opposite corners for boxes and ellipses. The label creation and control buttons are under the ‘Plot’ menu, whereas the other item creation and control buttons are under the ‘Plot Items’ menu.

For any of these items an initial parameter window comes up once the mouse button events have been received. Figure 8 shows the box creation window as an example. The other plot item windows have similar layouts with variations in the number of parameter entry fields. When the window appears one can then edit or set the values as needed before using the ‘Apply’ button. If one just closes the window, the parameters are left as indicated. One can change line colours and other properties, rotate the item, and change the exact values of the bounds for the item. Once an item is initially created it can be edited in a different type of window.

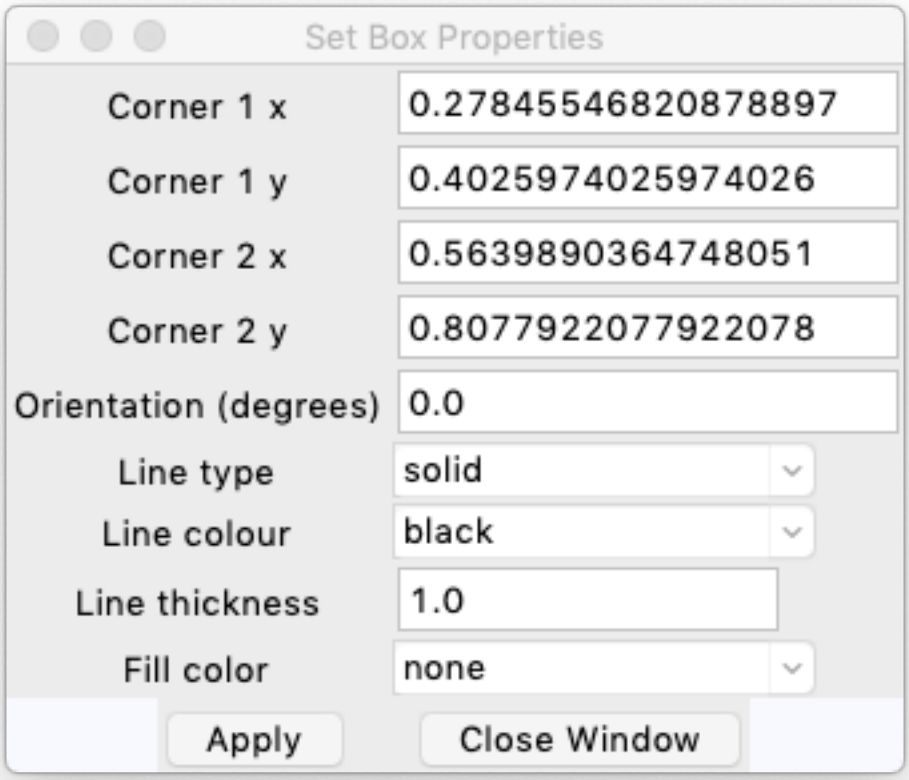

Figure 8: An example of a plot item creation window, for making a box in this instance. The line/vector/ellipse creation windows are similar in layout with some variation in the fields.

Editing Objects

For each type of object once the object has been created the parameters can be changed by editing the values. This brings up a different window than the one shown just above, as it is assumed to hold parameters for a number of objects at once. The different parameters are listed with tab characters separating the values. This is needed because various parameters might have spaces in them, such as the text of a label, and so one cannot use spaces as the separator. A tkinter scrolled text window comes up with the parameter values listed, and one can edit these as needed to change the object parameters.

Other Plot-Related Functions

The code provides the ability to make some auxiliary histogram plots related to a given main plot. The controls for both of these functions are found at left in the main window. The

options are to make a normal one-dimensional histogram of the points in the main plot or to make a two-dimensional histogram of the points in the main plot. The two-dimensional histogram is also called a density of points image. For a two-dimensional histogram one has to set a sampling size, the total number of values in each dimension (x, y). The code uses a set number of samples, the same for both x and y, so it produces a two-dimensional image of integer values, the number of points within the bin, of size N×N.

Once a two-dimensional histogram has been created one can store the numerical values, which are positive integers or zero, in an image file. This type of image is not like the PNG images that are used for pictures or for exporting the plots; it is stored in a format designed for storing a image as numbers, the Flexible Image Transport System or FITS. Such images are commonly used in astronomy. The code has a capability of reading such an image back in to examine so that if the user saves a two-dimensional histogram to a data file then they can read the values back in later.

For the regular one-dimensional histogram option, the points are collapsed in either x or y according to the radio button selection, and a histogram with a set number of bins over the plot range in either x or y is used for the plot. Histogram bars are used rather than individual symbols.

In this case an example will show the use of the functions. Assume that the following code is used to test the current numpy random number generator functions

```
import numpy
import math
from numpy.random import default_rng, SeedSequence
seed1 = 10254
# Use the new method to seed the numpy random number generator
rseed = SeedSequence(seed1)
gen1 = default_rng(rseed)
# generate sets of 1000 random numbers in the 0 to 1 interval, average these
# to a value, and accumulate 20000 of these for x and y values.  The output
# should closely approximate a Gaussian distribution in both values.
outvalues = gen1.random([1000, 20000])
meanvalues = numpy.sum(outvalues, axis=0)
xvalues = numpy.copy(meanvalues[0:10000])
yvalues = numpy.copy(meanvalues[10000:])
```

The xvalues and yvalues arrays are then used to make a plot. The prediction is that both x and y will have a mean value of 500.0, 1000 times the mean value of the uniform random number distribution, and that the standard deviation will be 9.128709175, the square root of 1000 divided by 12. Using the plot tool to make a plot with these values gives the display in Figure 9 below. The plot has been made square using the “Toggle Equal Aspect” function in the ‘Plot” menu, and the x and y axis ranges have been made the same for the plot.

With a cloud of data points of this type one can then check on the histograms and see what they look like, as well as calculating the data set statistics. One can set the number of histogram bins to 100, and plot the x histogram. This produces a new window as in Figure 10. If one produces a two-dimensional histogram with 100 bins or pixels the result is as in Figure 11. The code has the same number of samples in the x and y directions, making a square “image” in the display.

The actual mean and standard deviation values for the set x and y values can be found using the “Set Statistics” item in the ‘Sets’ menu. In this particular case the values are means in x and y of 500.039 and 499.94 with standard deviations of 8.98699 and 9.09409 respectively.

The standard deviations are a bit lower than expected in this particular trial. If one uses a much larger number of samples to estimate the mean and standard deviation the latter value approaches the expected value.

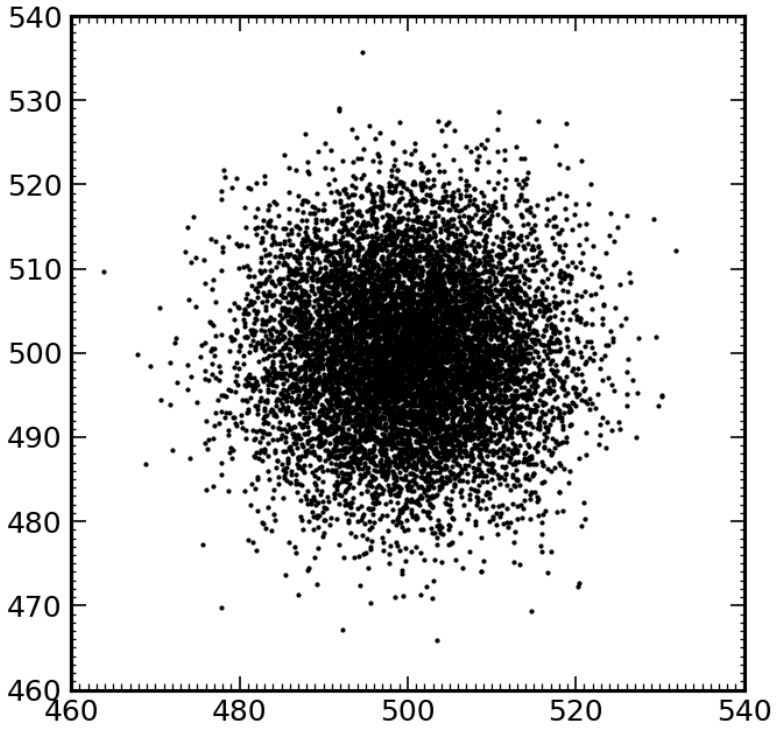

Figure 9: Results of a test of the numpy random number generator.

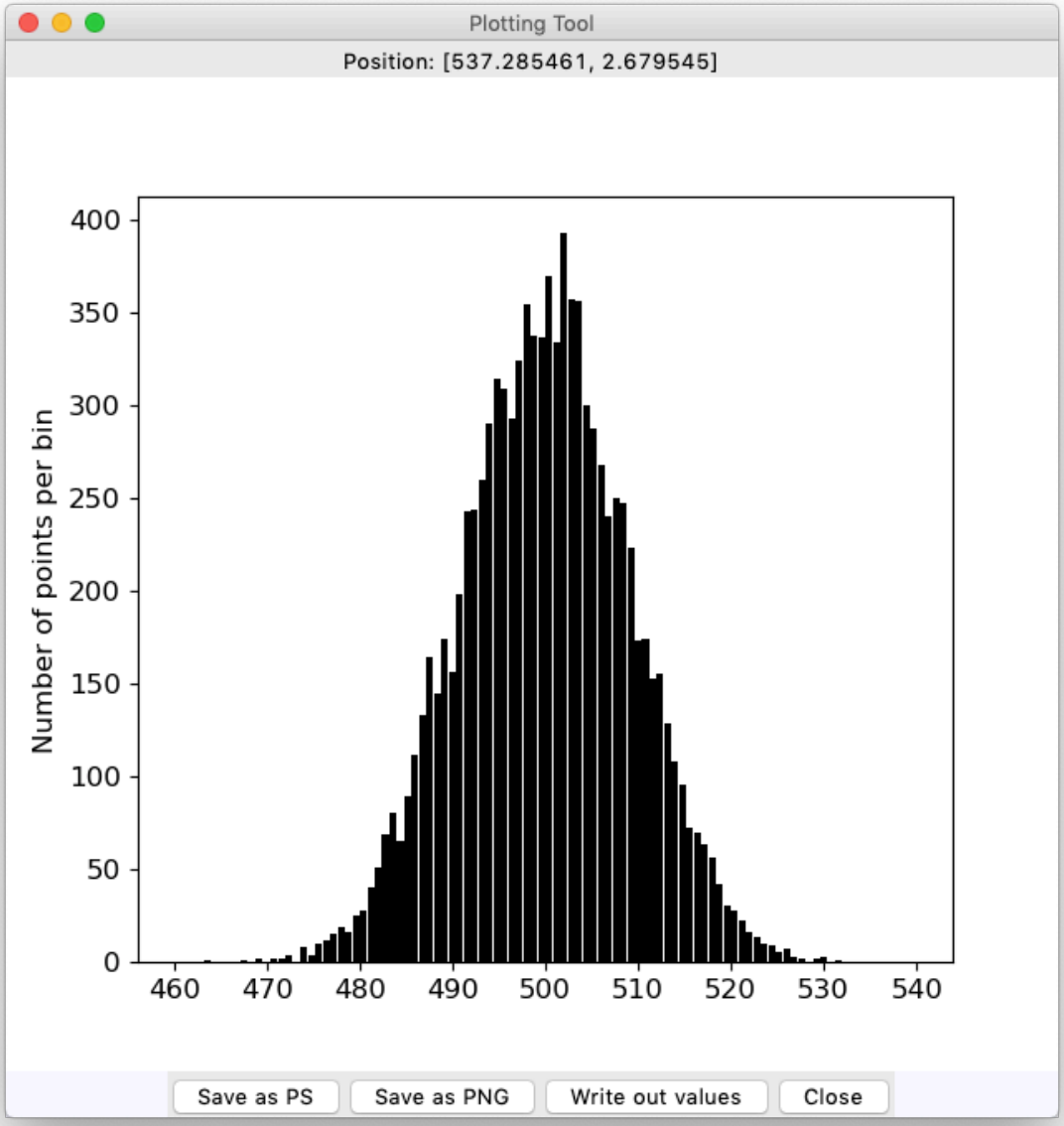

Figure 10: An example histogram, made from the data values shown in Figure 9.

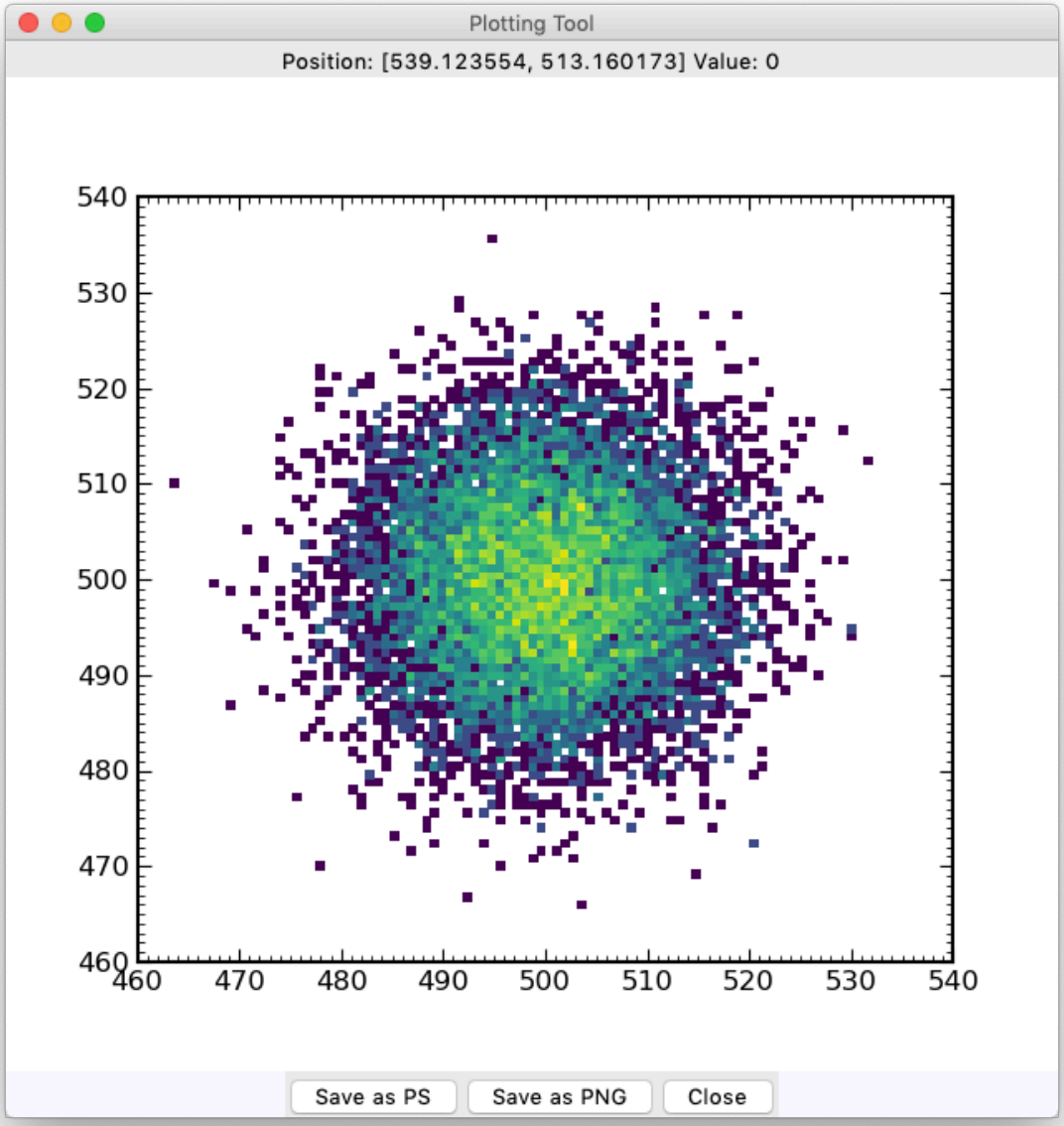

Figure 11: The two-dimensional histogram for the data points shown in Figure 9 above, with 100 samples in x and y.

Saving and Restoring Plots

The code has the option to save the internal state of the interface along with the data sets and the properties of the lines or other plot items to an ascii file. Such a file can then be read back into the code later preserving the plot. The ascii save file can be read and edited by a person, at least in principle. One probably needs to be cautious about doing this, unless one has some specific small parameter change one wants to make.

The code also has buttons at left in the main window to generate plot output files in either png or postscript form.