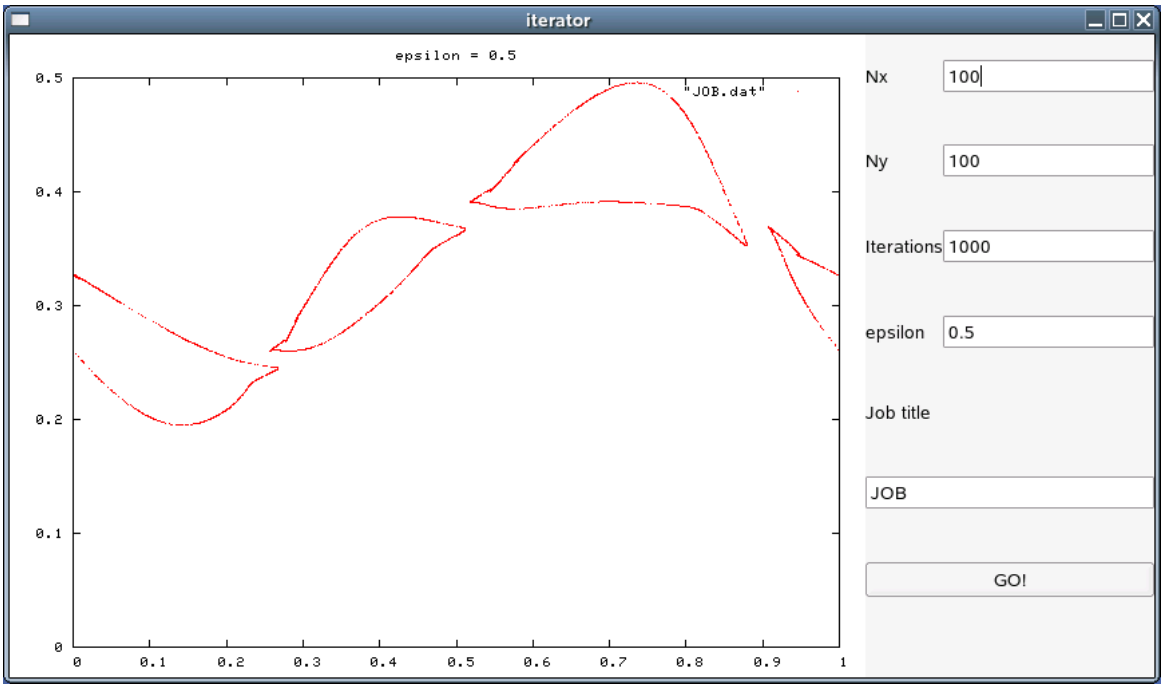
An introduction to GUI building with Glade

Bob Dowling

rjd4@cam.ac.uk

5 March 2007

Glade is a very simple GUI building package that runs on Linux (including PWF Linux) and Windows (subject to having the right Unix-compatibility layers added). This demonstration is given on PWF Linux. It consists of a worked example where I will take a set of existing command-line applications and convert them into a single GUI application that looks like this:



To get started, we will create a directory to work in and copy in the command-line application we want to use.

```
$ cd
$ mkdir  GladeCourse
$ cd  GladeCourse/
$ cp  /ux/Lessons/Glade/*  .
$ ls -l
total 14
-rwxr-xr-x  1 rjd4 rjd4 8842 2006-02-17 15:07 iterator
-rw-r--r--  1 rjd4 rjd4 2230 2006-02-17 15:07 iterator.c
-rw-r--r--  1 rjd4 rjd4   24 2006-02-17 15:07 Makefile
-rwxr-xr-x  1 rjd4 rjd4 1466 2006-02-17 15:07 template.py
$
```

This is everything we need for this course.

The programs

We are going to wrap our GUI around a simple fractal generator[1].

1 Purely for interest, the iteration is:

$$x' = x+y$$
$$y' = y(1 + \varepsilon \sin(2\pi x))$$

in the square $[0,1]\times[0,1]$, wrapped at the edges.

This represents the data generation program that you will have written or inherited that does your science. It takes a number of arguments on the command line that we want to set via our GUI. It generates a set of data points that need to be plotted.

```
$ ./iterator 100 100 1000 0.49 > 49.dat

$ ls -l
total 190
-rw-r--r--  1 rjd4 rjd4 180000 2006-02-17 15:11 49.dat
-rwxr-xr-x  1 rjd4 rjd4   8842 2006-02-17 15:07 iterator
-rw-r--r--  1 rjd4 rjd4   2230 2006-02-17 15:07 iterator.c
-rw-r--r--  1 rjd4 rjd4     24 2006-02-17 15:07 Makefile
-rwxr-xr-x  1 rjd4 rjd4   1466 2006-02-17 15:07 template.py

$
```

The `iterator` program takes four arguments. The first two set up how many points we start with, by defining an X by Y grid where the first two numbers identify X and Y. The third argument specifies how many iterations each point gets. The final argument is a parameter, ε, which defines the nature of the iterated function. It's typically this parameter we change most often. Part of our task in writing the GUI will be to make it easy to enter these four parameters without having to remember which order they come in and without having to retype values that don't change between runs.

The data now needs to be plotted. We will use a simple graphing program called `gnuplot` for this. Again, you don't need to know the details of `gnuplot`'s operation. The whole point of what we are doing is to hide all this. The details of `gnuplot` syntax are the sort of thing you get right once and never look at again.

```
$ gnuplot

        G N U P L O T
        Version 4.0 patchlevel 0
…
gnuplot> set terminal png
Terminal type set to 'png'
Options are 'small color picsize 640 480 '
gnuplot> set style data dots
gnuplot> set title "epsilon = 0.49"
gnuplot> set output "49.png"
gnuplot> plot "49.dat"
gnuplot> exit

$ ls -l
total 194
-rw-r--r--  1 rjd4 rjd4 180000 2006-02-17 15:11 49.dat
-rw-r--r--  1 rjd4 rjd4   4261 2006-02-17 15:13 49.png
-rwxr-xr-x  1 rjd4 rjd4   8842 2006-02-17 15:07 iterator
-rw-r--r--  1 rjd4 rjd4   2230 2006-02-17 15:07 iterator.c
-rw-r--r--  1 rjd4 rjd4     24 2006-02-17 15:07 Makefile
-rwxr-xr-x  1 rjd4 rjd4   1466 2006-02-17 15:07 template.py

$
```

Again, the details are mostly irrelevant to the task at hand, but what those `gnuplot` instructions meant was this:

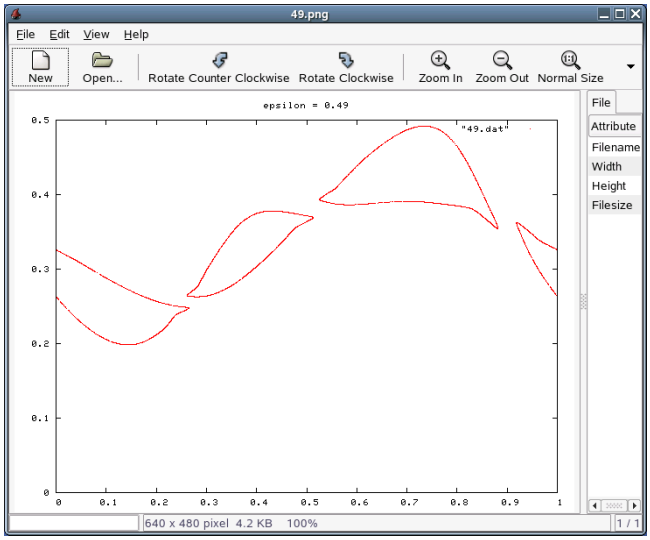
set terminal png	Define the output as being in PNG graphics format.
set style data dots	Specify that the individual points should be plotted and not joined together with lines.
set title "epsilon = 0.49"	Set the legend in the picture created.
set output "49.png"	Specify the graphics file created.
plot "49.dat"	Specify the input file and give the instruction to plot its data.
exit	Leave gnuplot.

What matters from our perspective is that we will want to get the user's parameter from the GUI into (at least) the title of the image created.

We can see the image created with the `eog` graphics viewer:

```
$ eog 49.png
```

which causes it to appear like this:



Note that there is no attempt to generate the graphics from the primary calculating program. It's generally bad practice to try to squeeze all the functionality into a single program. It's a much better idea to write programs to do just one task and then to combine these programs either in a shall script (for command line use) or in a GUI, which is what we will do here.

We will want the GUI to let us enter the four parameters and the file names used. We will work to the convention that if the GUI is given the name "`foo`" then the data file generated by the calculation should be "`foo.dat`" and the graphics file generated should be "`foo.png`".

Designing the GUI

First we must design the GUI. It is typically better to use a plain sheet of A4 paper to sketch in than any computerized system. We need to decide on the elements it must have and roughly where they should go.

We need the following:

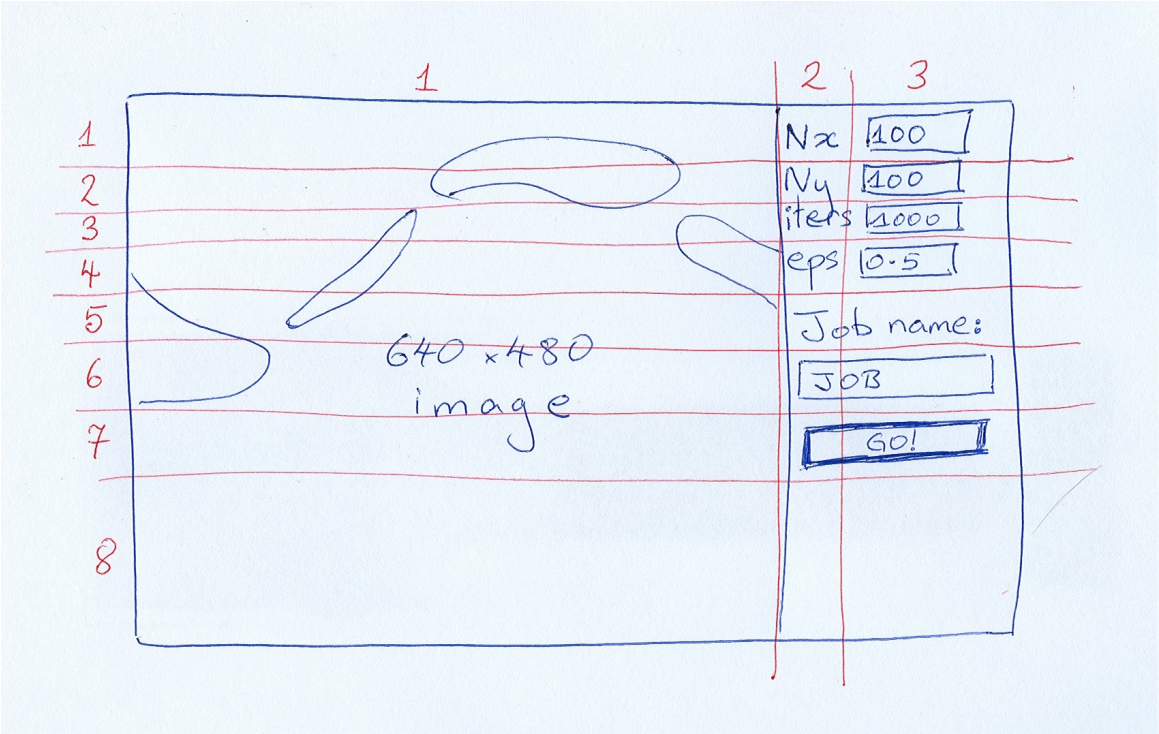
- number of elements in the X direction for the initial points,
- number of elements in the Y direction for the initial points,
- number of iterations to put each point through,
- value of epsilon,
- somewhere to display the resulting image, and

- somewhere to specify a file name or job name for storing the data.

My sketched design looks like this:

Note that it's hand drawn, with no attempt at beauty, artistry, or even straight lines.

The GUI builder works with a set of grids or tables possibly with one table sitting inside a single element of an outer table. So next I mark up my sketch with some construction lines (in red) to indicate how this table will work. Mine is a simple GUI so a single table suffices. Note that the rows and columns don't all have to be the same size.



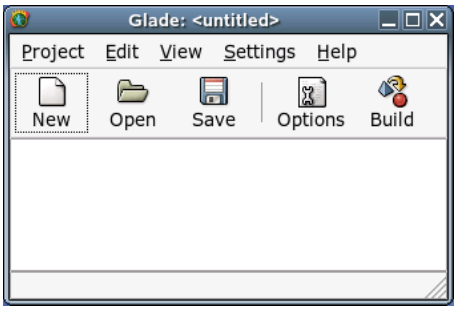
And that's the sum total of the design. Next I build it.

Building the GUI

First we must launch the program. The command to do this is `glade-2`. (There was a version 1 of glade which is incompatible with version 2. The command is named this way to make sure you never get version 1 by mistake.

```
$ glade-2 &
$
```

Launching Glade creates three windows: the main Glade window, a “properties” window and a “palette” window. We start with the main window and click “New” to start a new GUI building project. It asks us whether we want to build a “GTK+” project or a “GNOME project”. At the simple level where we will be working the difference is irrelevant. We will select “GTK+” because it is a little simpler and it doesn't make any real difference to us.



Once we make this selection the contents of the palette window stop being greyed out and are now available for us to use. The palette contains all the bits we build our GUI from.

If you move the mouse over the various items show in the palette a small pop up will give its name. These are the names we will be using throughout this course.

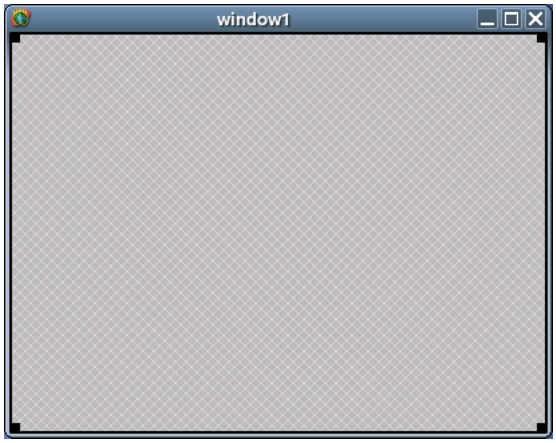
Main window

We need a main window to contain our GUI.

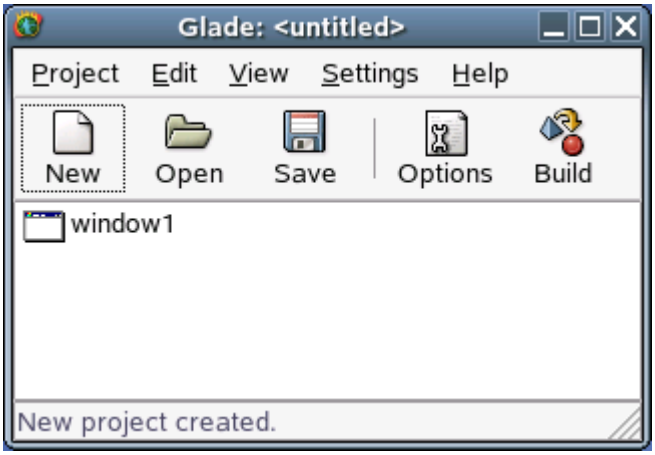
To do this, simply click on the window icon which is top left in the palette.

A number of things will happen.

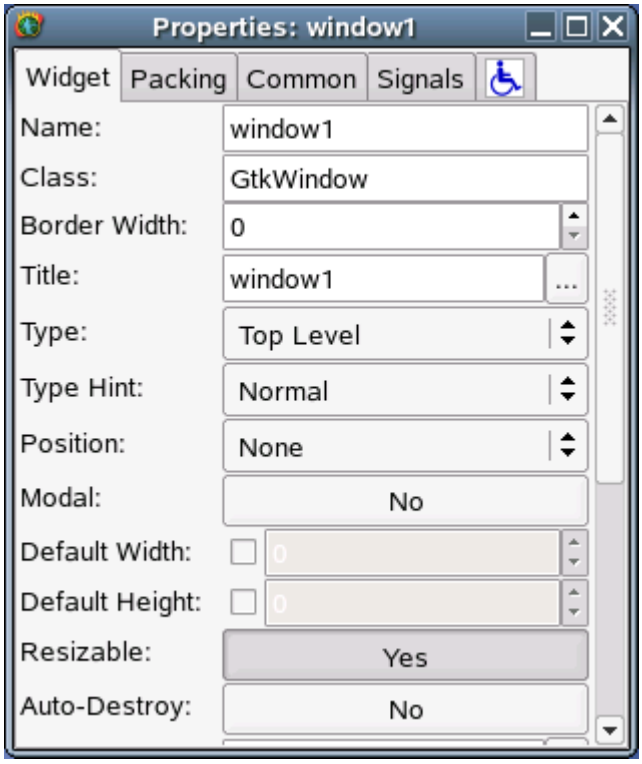
1. A new window with title “`window1`” will appear



2. The main window will gain a line listing window1.



3. The properties window will stop being greyed out and will show the mostly non-existent properties of window1, with the “Widget” tab active.



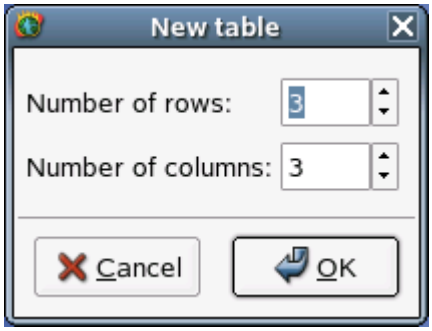
We will only address the properties that we plan to change. At this point, the only property we want to change is the window's title from the rather bland "`window1`" to something that reflects our application: "`iterator`". We will change the text in the properties window corresponding to the "Title:" entry. There's no need to hit Return, though it won't do any harm. As you type you should see the title of the window change before your eyes.

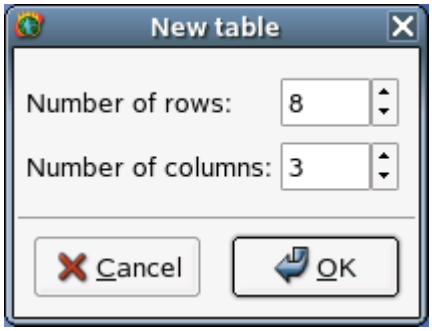
Splitting the window

Next we need to split the window into the cells that will be occupied with the various components of the GUI. According to our diagram we need three columns (with the first much wider than the other two) and eight rows.

To get a table first click on the table icon in the palette to say that we want a table. Then click into the window to say where we want it.

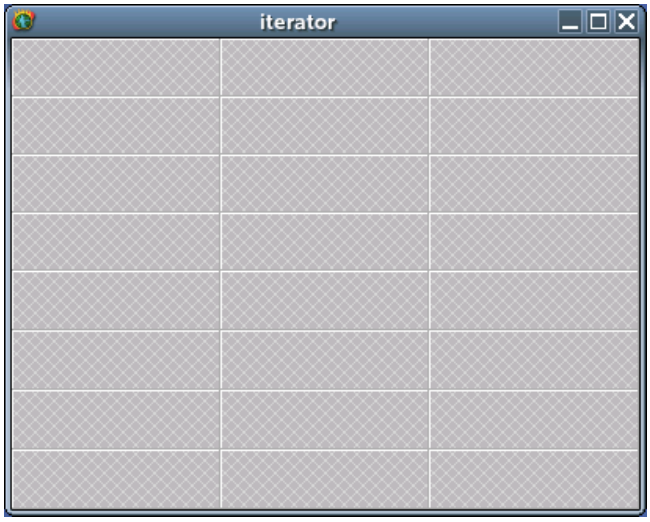
A dialogue box asking for details of the table will appear. It defaults to having three rows and three columns. Change this to eight rows and three columns. Then click OK.





The GUI building window will develop a 8×3 table and the properties window will change to

describe the table's properties.



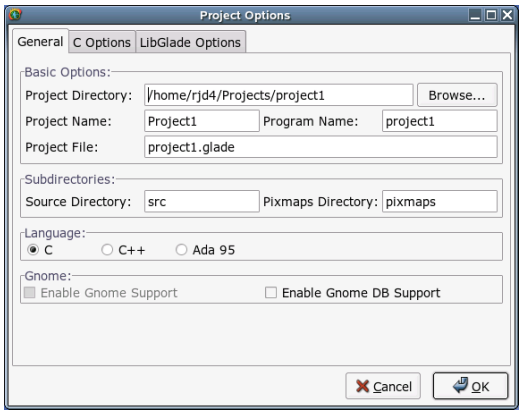
Now all we have to do is to add items to the table, remembering that the image has to span all eight rows and the first column will have to be very wide.

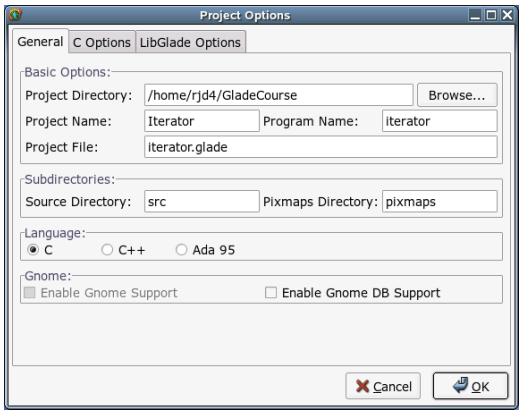
Saving our work

It's important to remember to save work regularly. This applies as much to GUI development as it does to writing dissertations. Click the “Save” icon on the main window.

Instead of just a normal “save as...” dialogue this actually launches an “options” window. Don't be worried; the top of the options window sets the names of the files and the directories. Change the directory to be `GladeCourse`. You will find that quite a few other file names will change in sync. In this example I am giving my project the name “`Iterator`”.

Click “OK” to confirm.



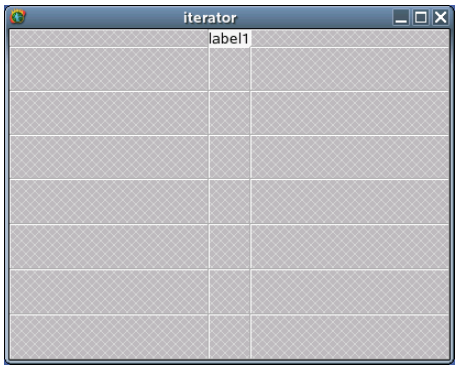


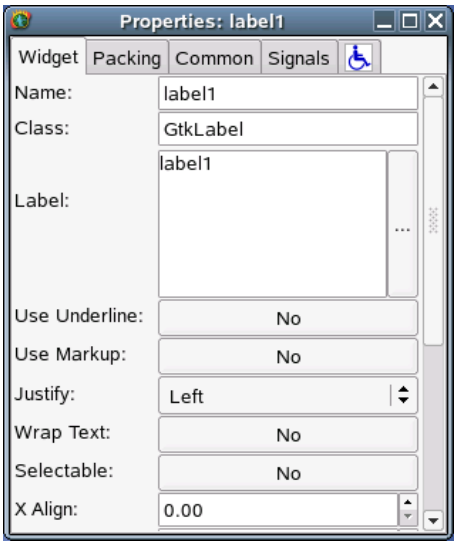
Adding the labels

We will start with the easy ones. In the second column, the top four table cells need to have simple text entries added to them, “Nx”, “Ny”, “Iters.” and “epsilon”. These are called “labels”.

To add the “Nx” label, click the label icon in the palette and then the top middle cell of the table.

Two things happen. The window changes to reflect the addition of a new label and the properties window shows the properties of the new label.

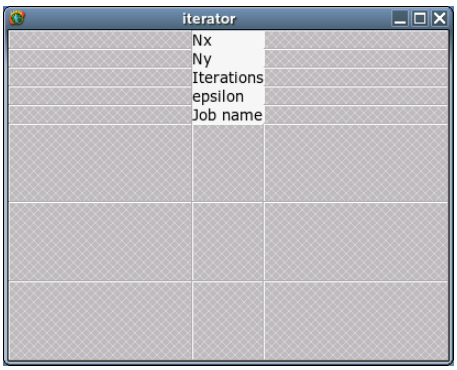




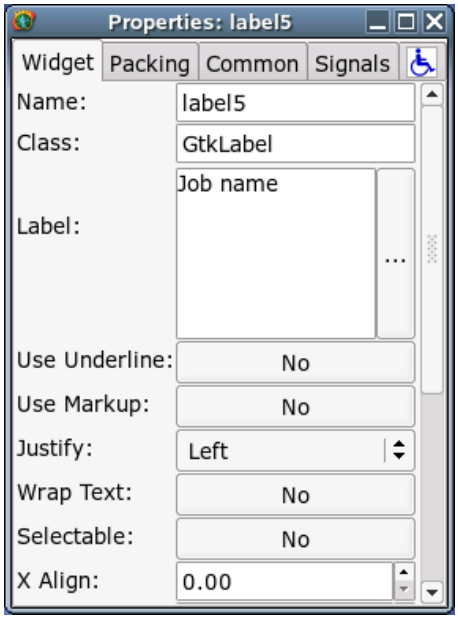
We are going to change one property of this label: the text of the label itself. The properties window is showing a text area labelled “Label:”. We will change that to the text we want in the label: “Nx”. Again, there is no need to hit Return, and if you do you will just add a line into the label itself.

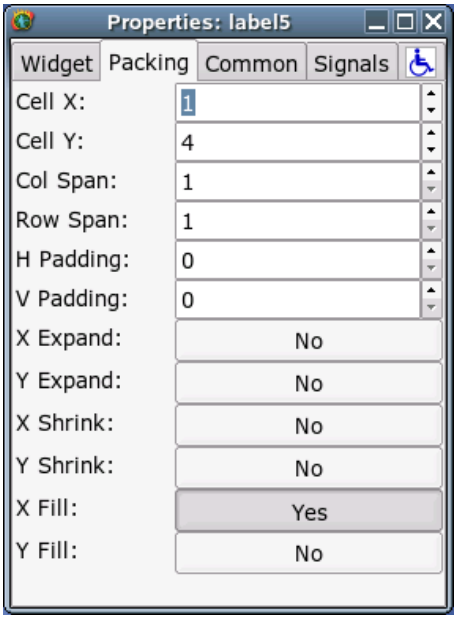
Next we do the same for “Ny”, “iterations” and “epsilon”.

Finally among the labels we have to create one reading “Job name:” for the file saving. This label has to span two columns. We start by adding the label to the left hand cell of the pair. (If it was spanning rows as well as columns we would add it to the top left cell of the set.)

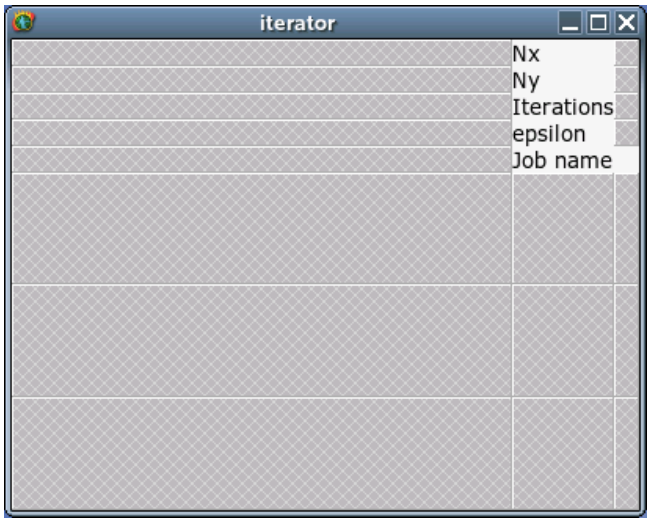


To change it to span two columns we turn our attention to the properties window and switch tab from “Widget” to “Packing”.





In the data under the “Packing” tab we see an entry “Col Span:”. This dictates how many columns the item span and we will change this to 2. We do have to hit Return for these entries. The GUI window lurches over a bit as the table's cells resize, but we will deal with that later. For the time being we have a label spanning columns 2 and 3.



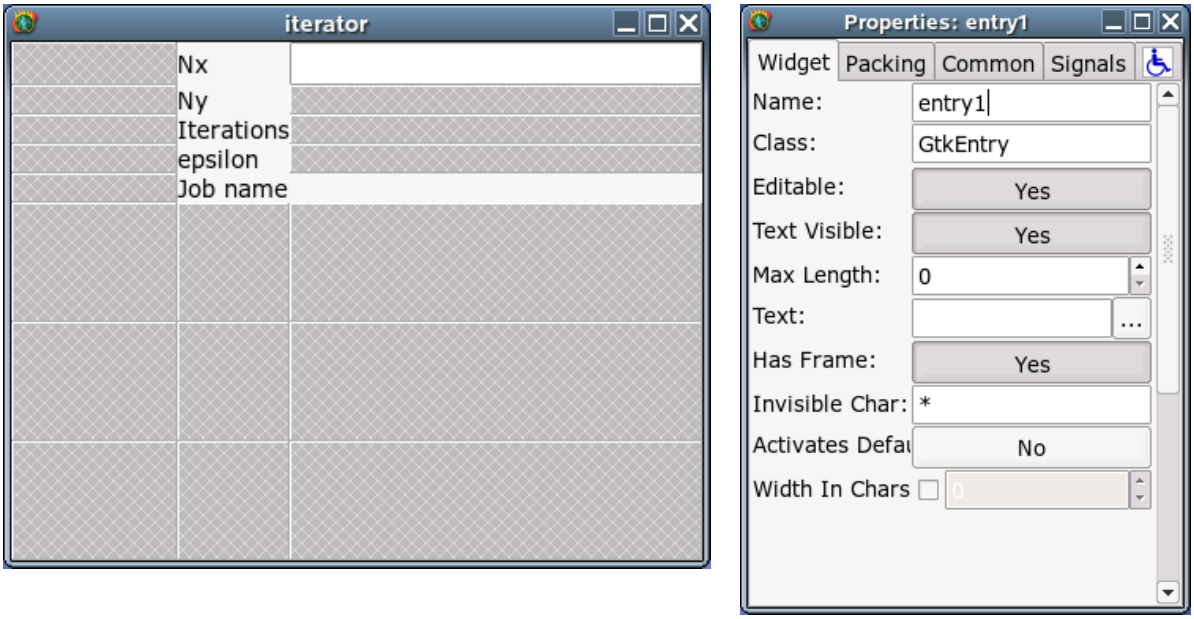
And that completes our labels.

Text entry

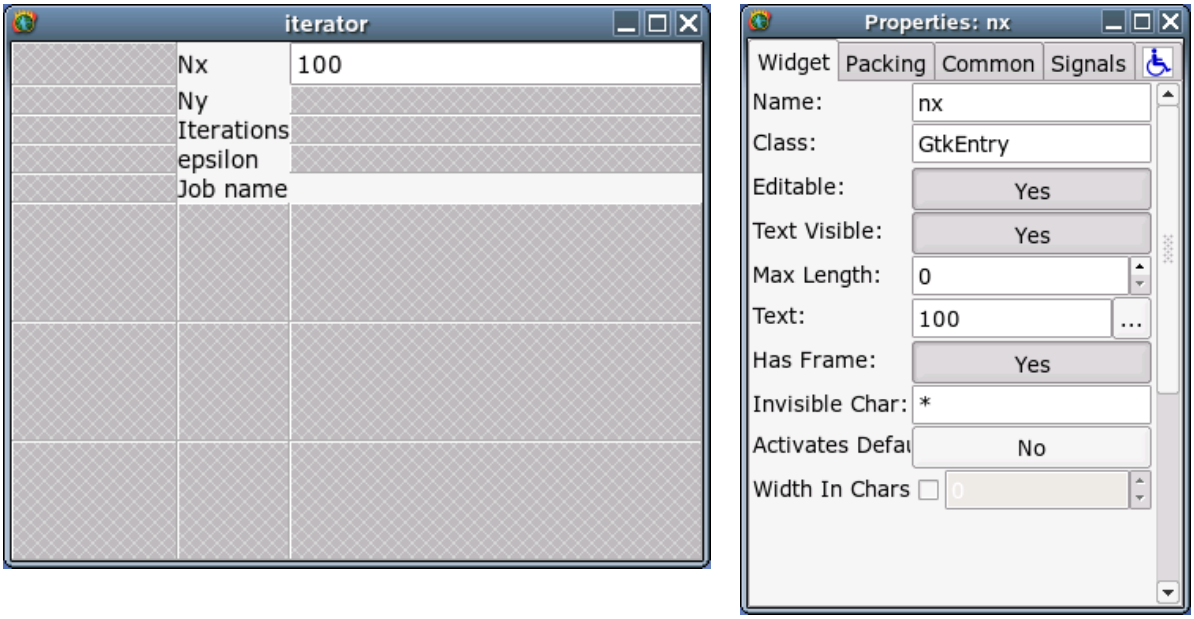
For each of these labels we need an area where the user types in some data. We need integer values for Nx, Ny and iterations, a floating point number for epsilon, and some text for job names.

For this we use a “text entry” widget.

As usual, we click on the text entry icon in the palette and then in the cell where we want to place it. We start with the text entry next to the “Nx” label.

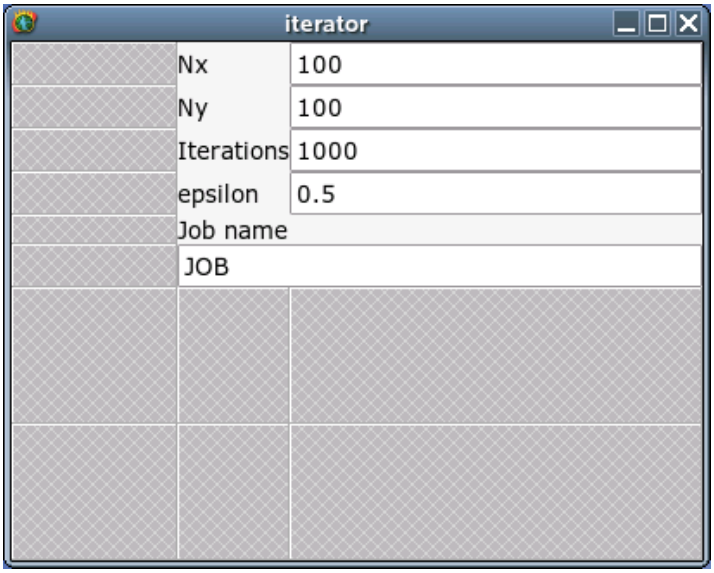


This text entry widget starts with the name "`entry1`". Up to now we have not bothered changing any widgets' names because we won't have to deal with them directly and don't really care what they're called. These widgets we will interact with because we want to suck data out of them. So we'll give them sensible names. We'll call this one "`nx`" by replacing "entry1" with "nx" in the "Name:" field. We also have to give it some initial text which we enter into the "Text:" field. We'll use a default value of 100.



We will repeat this process for "`ny`", "`iters`", "`epsilon`" and "`jobname`". The job name text entry widget can be spanned over two columns in exactly the same way as the label was. We will use some default values to help the user:

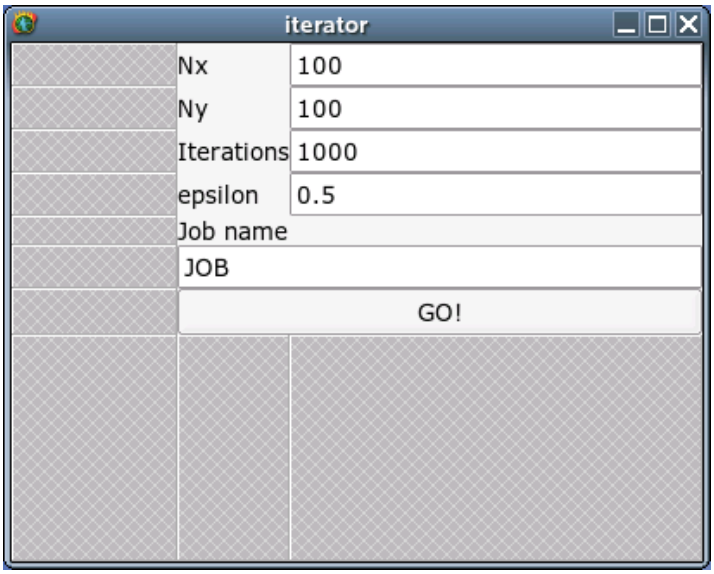
Nx, Ny	100
iterations	1000
epsilon	0.5
jobname	JOB



Buttons

Next we add the “GO!” button.

As ever, click on the button icon in the palette and then in the table cell where you want it to go. It can be made to span two columns by using the “Packing” tab in the properties window. Again, we will give it a proper name: “go” and change its text to be the button's “GO!”.



There's a lot more we can do to clean it up, but we will do that later when everything is in

place.

The image

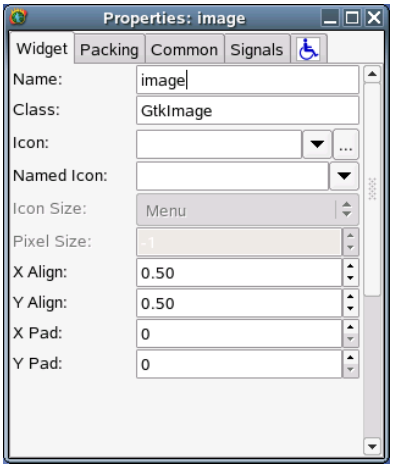
Now we will add the image viewer. This needs to span all eight rows and to occupy the entire first column. The `gnuplot` program is generating 640×480 images so we should size for that.

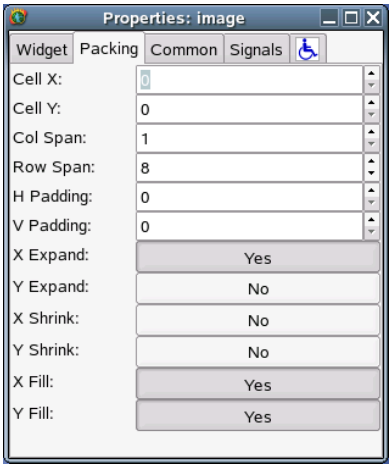
Add an image by clicking on the image icon in the palette and then in the top left cell of the table. Note that we need to select “image” and not “drawing area”. The latter is for people to create their own drawings not to display existing ones.

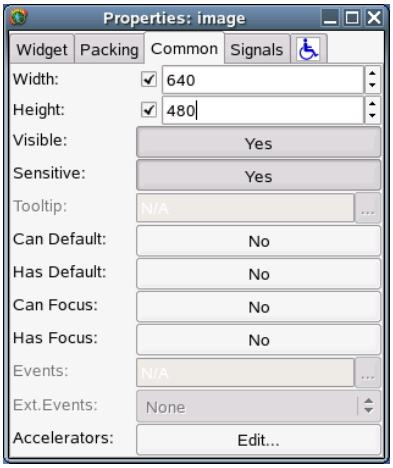
We will give it the name “`image`” in the “Widget” tab in the properties window.

We specify that it spans eight rows in the “Packing” tab.

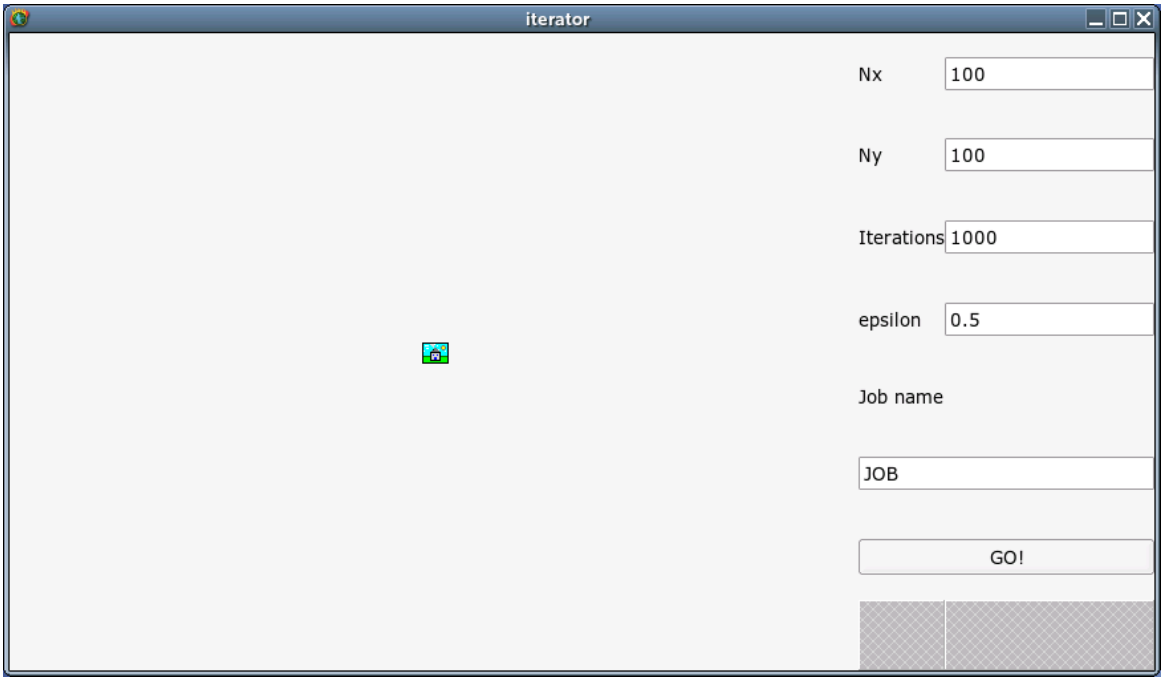
Finally, we move to the “Common” tab and activate the fields that let us explicitly set the width and height of the image. These need to be enabled by the check boxes and then have values added. Because most widgets resize these fields are normally deactivated which is why we need to explicitly activate them.







The main window is now quite enormous. Fortunately we are almost done with it for now.



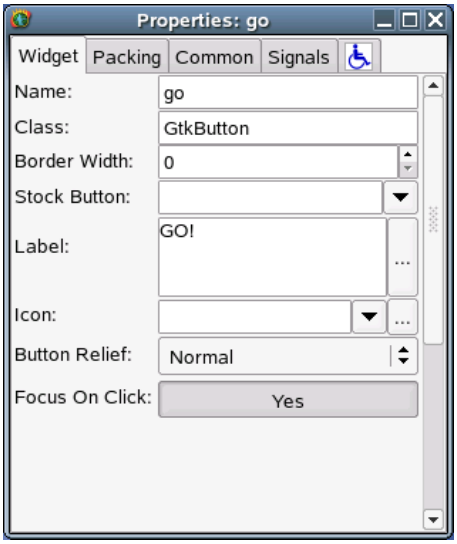
Events

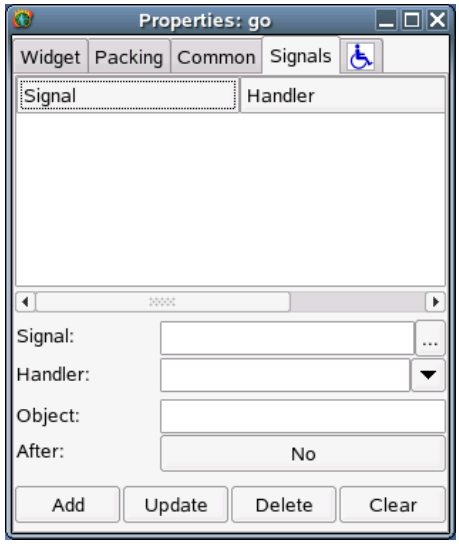
A GUI is an “event-driven” application. When you click a button an “event” is generated saying you have done so, for example.

To get our application to work we need to do something when the “GO!” button is pressed. The way to do that in Glade is to use the properties window on the button to identify a function in a script when the button is clicked. Then, separately, we write the script that implements that function.

The go button

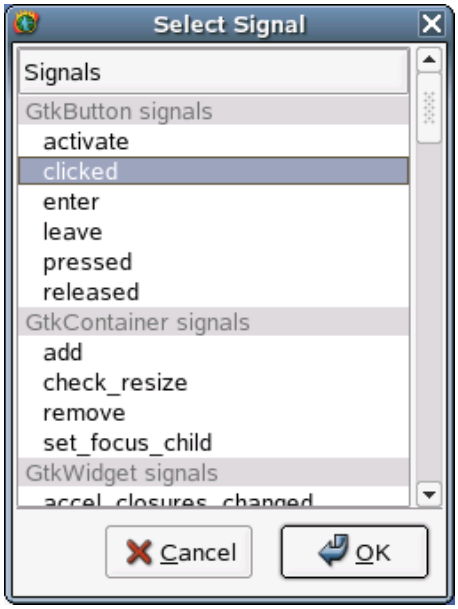
To set up the signal handling for the “GO!” button we first click on it in the GUI to select it. The properties window will switch to showing its properties. This time, select the “Signals” tab.





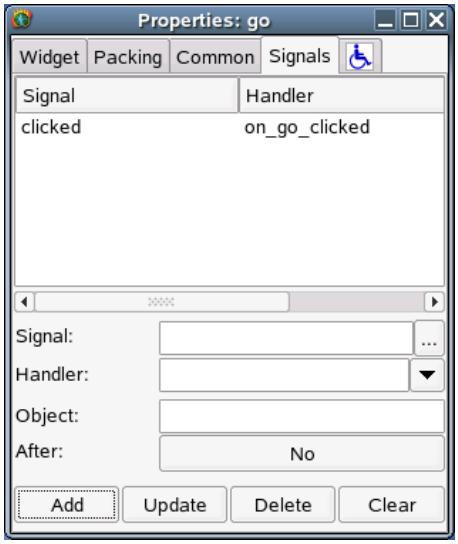
Every widget generates certain events or signals[2]. Some are common over all widgets, others specific to a specific type of widget, and almost all are irrelevant to us. We are interested in only one event which is called “clicked”. This is the event generated by the button when we click it.

To identify that event we can either type “`clicked`” in the “Signal:” field, or select it from the browsable list under “…”.



Note that as soon as the “clicked” event is selected, a suggested name for the “handler” is filled in: “`on_go_clicked`”. This is a suggested name for a function that will be called **on** the event of the **go** button being **clicked**. We do not have to stick with this proposed name but we are taking the path of least resistance in this course so we will. Click “Add” to approve this assignment of handler function to event. The properties window will update to list this assignment in the top part of the window.

2 This is not a good name for them. They should not be confused with the signals used by the operating system to stop or start processes etc.



Of course, we still have to write this function.

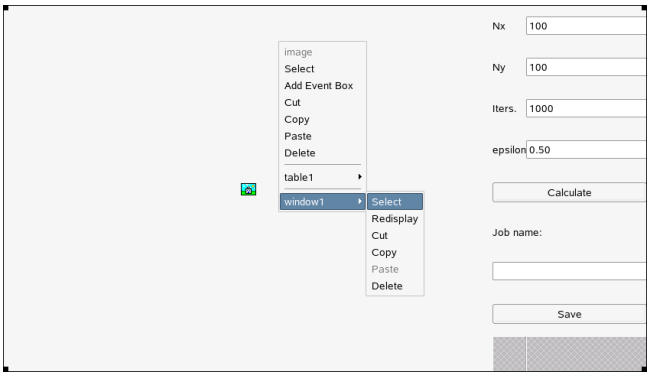
Closing the window

When we close the main window from the buttons in the title bar we expect the application to terminate. Surprisingly this is not done by default. We need to set up an event handler to handle the event sent to the application when its window is closed.

To add an event handler we need to select `window1` so that its properties are available and so that we can access its “Signals” tab. There is an interesting twist, though. We have completely covered `window1` with a table. That table is mostly covered by other widgets. There is no part of the main GUI that gets us the underlying `window1`.

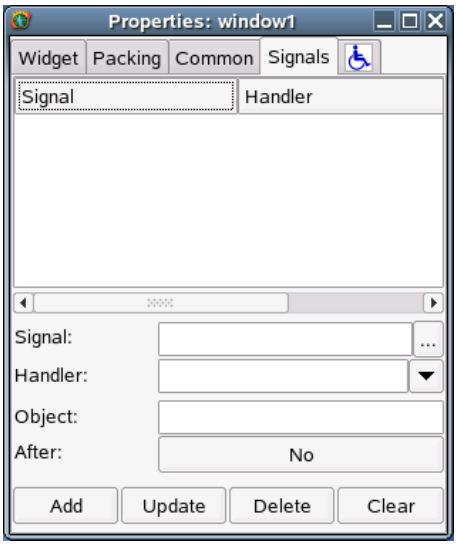
There are two ways round this. We could just click on the `window1` line in the main GUI builder window. But this would not be available to us if we were trying ot get at some other inaccessible widget.

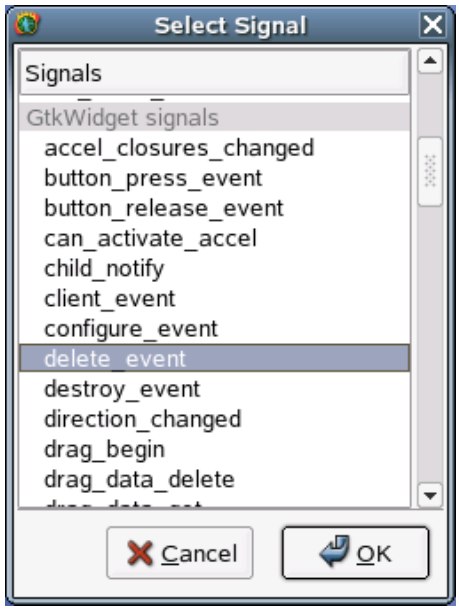
The more general approach is this. If you click on any part of the window representing our application with the *right* button on your mouse, you get a menu. This menu lists options for whatever the widget is that you were clicking on. However, beneath that it provides submenus for all the widgets below the top-most widget. In our case the bottom of the menu is “`window1`”. Moving across from that we get its menu options, the first of which is “Select”. If we pick this we have selected the concealed `window1` widget and can address its properties in the Properties window.



```
#!/usr/bin/python

import sys
import os
import gtk
import gtk.glade

# Put the event handlers here.
def on_window1_delete_event(*args):
    sys.exit(0)

def on_thing_clicked(*args):

    # Get values out of the data input widgets
    foo_value = foo_widget.get_text()

    # Run the calculation command.
    calculation_command = "PROGRAM '%s' > '%s.dat'" % (foo_value, jobname_value)
    os.system(calculation_command)

    # Generate the gnuplot commands
    gnuplot_filename = "%s.gplt" % jobname_value
    gnuplot_file = open(gnuplot_filename, "w")
    gnuplot_file.write("""
set style data dots
set title "foo = %s"
set terminal png
set output "%s.png"
plot "%s.dat"
""" % (foo_value, jobname_value, jobname_value)
                        )
    gnuplot_file.close()

    # Run the gnuplot command
    gnuplot_command = "gnuplot %s.gplt" % jobname_value
    os.system(gnuplot_command)

    # Take the image file created and stick it in the image widget
    image_filename = "%s.png" % jobname_value
    image_widget.set_from_file(image_filename)

# Main function.
if __name__ == "__main__" :
    # Put the name of the .glade file here.
    global application
    application = gtk.glade.XML('thing.glade')

    # Create the widgets we need to talk to directly here.
    global image_widget
    image_widget      = application.get_widget('image')

    global foo_widget
    foo_widget        = application.get_widget('foo')

    # Launch the application
    application.signal_autoconnect(globals())
    gtk.main()
```

Again we pick the “Signals” tab and identify the “delete_event” from the “Select Signal” browser.





Again, we will accept the proposed name “`on_window1_delete_event`” for the function.

We are now done with the GUI building. Save the interface and then quit. (Go to the main window and select Project → Quit from the menus.)

Writing the back end script

The Glade system can have various languages running behind the GUI, including C, C++ and Ada. It can also interface with the Python scripting language, which is relatively easy to learn and quick to experiment with. Moreover we can use a template script for the task of sticking a GUI on the front of an existing command line application so we only need to make simple changes rather than writing it from scratch.

First we must be in the directory identified in the original “save as...” dialogue. In the case of this example the directory is `/home/rjd4/Projects/Iterator`.

We will copy a template Python script that we will then modify. We will call it “`iterator.py`”. Our editor of choice will be `emacs` in this set of notes. There is no obligation to use this editor; any plain text editor will do but `emacs` has nice support for editing Python files.

```
$ cp template.py iterator.py
$ emacs iterator.py &
$
```

This course does not have a Python prerequisite. You are not expected to follow all the ins and outs of the Python in the script. You just need to follow the patterns that already appear in the script and change them and duplicate them as needed.

The text that follows is the entire template. The elements you have to change are highlighted. While the script may look overwhelming at first glance, there are just *six* entries in this script that need to be modified. Each is individually quite straightforward and they will be covered one by one in the following sections.

In the words of the late, great Douglas Adams: **Don't Panic!**

It's easier to understand the changes to the template script if we don't tackle them in the

order they appear but in a slightly different order.

The name of the Glade file

```
    application = gtk.glade.XML('thing.glade')
```

We will start with one of the shortest changes. The Python script needs to know where Glade has put its description of what the GUI should look like. In the current directory, you will find a file ending in ".glade":

```
$ ls -l
total 28
-rw-r--r--  1 rjd4 rjd4 11183 2006-02-16 20:27 iterator.glade
-rw-r--r--  1 rjd4 rjd4  1285 2006-02-16 20:02 iterator.glade.bak
-rw-r--r--  1 rjd4 rjd4   277 2006-02-16 20:27 iterator.gladep
-rw-r--r--  1 rjd4 rjd4   277 2006-02-16 20:02 iterator.gladep.bak
-rwxr-xr-x  1 rjd4 rjd4  1587 2006-02-16 21:00 iterator.py

$
```

We change "thing.glade" to "iterator.glade":

```
    application = gtk.glade.XML('iterator.glade')
```

The widgets

```
    global foo_widget
    foo_widget         = application.get_widget('foo')
```

This part of the script lists the widgets that we have to interact with directly. In our case this is the set of widgets we will be pulling parameters from ("nx", "ny", "iterations", "epsilon", and "jobname") and the image widget. The image widget is always needed so we can just leave the line alone.

So we will replace the single "foo" entry in the script with four entries for each of our inputs. We will stick to the naming convention in the template of using "foo_widget" as the reference to the widget that was given name "foo" in the GUI builder.

So the "global" line should be altered to read:

```
    global nx_widget
    global ny_widget
    global iterations_widget
    global epsilon_widget
    global jobname_widget
```

and we will need five equivalent "foo_widget" lines:

```
    nx_widget          = application.get_widget('nx')
    ny_widget          = application.get_widget('ny')
    iterations_widget  = application.get_widget('iterations')
    epsilon_widget     = application.get_widget('epsilon')
    jobname_widget     = application.get_widget('jobname')
```

These set up the variables used in the script and, if you're interested, the “global” keyword simply means they can be used anywhere in the script.

The name of the event handler

```
def on_thing_clicked(*args):
```

You will recall that we used the GUI to name a “handler” function for the “clicked” event on the “go” button. The GUI offered the proposed function name “on_go_clicked” which we accepted. We commented then that we still had to write the function. This is where we do it. The “`def`” keyword simply means “define a function” and the name that follows it is the name of the function being defined.

So we change this line to read

```
def on_go_clicked(*args):
```

matching the function name expected by the button.

The “`(*args)`” bit at the end is simply the language's way of describing how information is passed to the function and the colon just marks the end of the naming and the start of the definition.

The event handler function immediately above it (`on_window1_delete_event`) is the event handler called when the window is closed (“deleted” in X11 jargon).

Getting values from the text entry widgets

```
    # Get values out of the data input widgets
    foo_value = foo_widget.get_text()
```

The `foo_widget` is any of the widgets created in the section at the bottom of the script. We can talk about them directly because they were declared to be global. All of the widgets are text entry widgets and the `foo_widget.get_text()` instruction simply gets the text they are currently displaying. It always returns text, though. To a computer the text “100” is very different from the number “100” but this won't matter to us because of how we are calling the command. Whatever this text is, it is put in a variable `foo_value` for later use.

So to start the work for the nx widget we would use the line

```
    nx_value = nx_widget.get_text()
```

So the line

```
    foo_value = foo_widget.get_text()
```

in the template would be replaced with the five lines

```
    nx_value         = nx_widget.get_text()
    ny_value         = ny_widget.get_text()
    iterations_value = iterations_widget.get_text()
    epsilon_value    = epsilon_widget.get_text()
    jobname_value    = jobname_widget.get_text()
```

in the real script.

Running the calculation

```
    calculation_command = "PROGRAM '%s' > '%s.dat'" % (foo_value, jobname_value)
```

This is the line where we specify the command line we want run behind the scenes to create the data that we are then going to display in a graph.

We need to define the program to be run ("PROGRAM" in our template) and its command line arguments which we populate from the data we have pulled from the GUIs. In our case we have to replace "PROGRAM" with "./iterator" because that's the program we want to run. It's "./" at the start to mean "run the iterator program out of this directory". Normally the current directory wouldn't be looked in for programs.

We also need to define its command line arguments. All the business with % characters is Python's way of doing substitution of text. If foo_value was "100" and jobname_value was "fred" then Python turns

```
"PROGRAM '%s' > '%s.dat'" % (foo_value, jobname_value)
```
into

```
"PROGRAM '100' > 'fred.dat'"
```
before setting this converted form to be the value of calculation_command. The %s expressions get converted into the values of the variables appearing in the brackets at the end in the order they appear.

Our ./iterator command takes four variables: Nx, Ny, iterations and epsilon. So we need four %s expressions for the arguments and a fifth for the stem of the file name. So for our script we need the line

```
    calculation_command = "./iterator '%s' '%s' '%s' '%s' > '%s.dat'" %
        (nx_value, ny_value, iterations_value, epsilon_value, jobname_value)
```

in place of the template's line. *This must all be on a single line.* So, if nx_value was "100", ny_value was "200", iterations_value was "1000", epsilon_value was "0.49" and jobname_value was "run49" then this line would be converted to

```
    calculation_command = "./iterator '100' '200' '1000' '0.49' > 'run49.dat'"
```
before being run.

Converting the graphics

We use the gnuplot application to convert our data points into a graph. Now gnuplot is a huge application in and of itself and this is not a course on it. This section will lead you through the gnuplot instructions and offer a few alternatives that you might need for different forms of graph plotting.

The gnuplot instructions in the template include these that you might want to change:

```
set style data dots
set title "foo = %s"
```

We will explain them both, but note that we are using the %s mechanism explained in the section above so we will have to change the variable names in the brackets at the end.

```
set style data dots
```

This command specifies that the data consists of individual data points that should not be joined together with lines. The alternative, if you want the points joined by straight lines is "set style data lines".

```
set title "foo = %s"
```

This sets a title for the graph which appears at the top of the image by default. If you don't want it then just drop the line altogether and remove the `foo_value` variable from the brackets. We want to quote the value of epsilon in our title so we would change this line to

```
set title "epsilon = %s"
```

and change the “`foo_value`” in brackets at the end to “`epsilon_value`”:

```
""" % (epsilon_value, jobname_value, jobname_value)
```

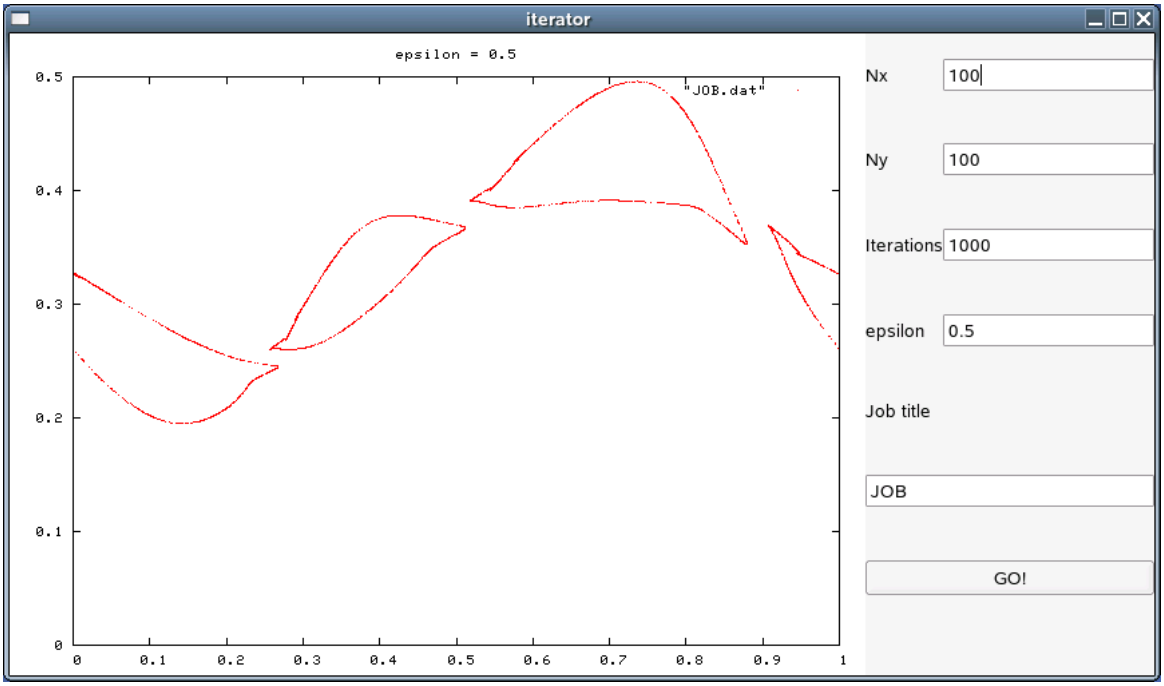
And that's it. We have modified our template script to process our specific data. Save the edited Python file (Ctrl+X Ctrl+S) and we are ready to run our GUI.

Running the GUI

To launch the GUI, simply run the Python script.

```
$ ./iterator.py
```

Any error messages will appear in the terminal you launched the program from. The GUI should appear in front of you. Click “GO!” to see how the defaults play out.



You can close down the Glade GUI builder if you haven't already. Go to the main window and select Project → Quit from the menus.

Exercises

Resetting default values

Relaunch Glade in the `GladeCourse` directory. Double click on `window1` in the main window to bring up the window you use for building the GUI.

Now add a button spanning the two remaining cells on the bottom row. This button should carry the text “Defaults” and should set the values of the text entry widgets to the values we started them out with.

You have everything you need to do this with the exception of the small tit-bit of information that as well as `foo_widget.get_text()` to *get* the text from the widget we can use `foo_widget.set_text("`*`text`*`")` to *set* the text in the widget.

Prettifying the GUI

Access the properties of the table and increase its number of rows by one. Increase the image's row span to occupy this extra row. Then add a blank label spanning the second and third columns on this new row. Access the packing properties of the label and switch the “Y Expand” property from false to true. Observe how the rows are no longer stretched so badly.

Same ideas, different program

The `lissajou` program takes three arguments: m, n and delta. It generates 1,001 lines of output representing the coordinates of points

$x = \sin(2\pi\ mk/N)$

$y = \sin(2\pi\ nk/N + 2\pi\delta)$

$N = 1000$

$k=0...1000$

Write the GUI and adapt the template to create a program that inputs the three values and a job name and displays the image of the corresponding Lissajou figure.

The `gnuplot` instructions you will need can be adapted from this set for m=3, n=4, delta=¼:

```
set title "Lissajou: m=3 n=4 delta=0.25"
set terminal png
set output "test.png"
plot "test.dat"
```