GrapherGUI Tutorial

Bo Shi
Ocean Engineering, Vortical flow lab

bshi@mit.edu

GrapherGUI is a Java library that allows you to quickly develop graphing utilities. You provide the mathematical functions and the parameters, and the user interface is handled by the library. With only a few lines of code, a Java application or Applet like the one shown below can be created. This document is designed to be a quick tutorial and introductory document to the GrapherGUI package. For more detailed documentation, refer to the source documentation (../index.html).

1. A Simple Example

1.1. Step by Step

The following steps describes the process of creating a graphing tool using GrapherGUI. In the directory where the GrapherGUI library is located, create a file `classname.java` where `classname` is an arbitrary name for your application.

1. Create a `SliderMatrix` object.
2. Create up to ten classes which extend the `Function` class.
3. Create a `GUI` object.
4. Add the `GUI` object to the Applet

1.2. A Complete Program

Web.java creates an Applet. To create a standalone program that does not require a browser to run, minor changes are required. Consult this section. The source code to Web.java can be found here (Web.java.html). Following is a description of the important lines of code.

To use the classes provided in the GrapherGUI library, include the package.

```
import GrapherGUI.*;
```

The class that we have here is called `Web`. The naming of the class is arbitary. Because we wish to compile this program as an Applet, our main class must `extend Applet`. When a class extends from Applet, the `init()` method is required. This is where all the work will be done. Within `init()`, we need to create instances of the objects described in the previous section.

```
public class Web extends Applet {
    public void init() {
  // ... insert code here ...
    }
}
```

If you wish to use parameters in your functions, declare a `SliderMatrix` object. You can add as many parameters as you need and any given function does not have to use all (or any) of the existing parameters. Internally, `SliderMatrix` keeps a list of `ParameterPanel` objects. The following is an image of what a `ParameterPanel` looks like:

The constructor for `SliderMatrix` takes an integer which will be the maximum number of parameters that can be added. In order to make a parameter available to your functions, you need to use the `add()` method in the `SliderMatrix` class. `add()` takes six parameters:

1. Minumum parameter value
2. Maximum parameter value
3. Starting value
4. Number of separators on the slider bar
5. Filename of the image file associated with the parameter (this can be null)
6. String name associated with parameter (this will be displayed if the image file is null or cannot be found)

Remember that you may only use JPEF, GIF, and PNG (on newer Java VMs). The path to the images is relative to the directory where your main JAR or .class file is located.

```
SliderMatrix sm = new SliderMatrix(3);

sm.add(-10.0, 10.0, 0.0, 4, "param_h.jpg", "h");
sm.add(5.0, 15.0, 5.0, 4, "param_g.jpg", "g");
```

... declaring functions here

```
MyFunc1 f1 = new MyFunc1();
MyFunc2 f2 = new MyFunc2();
```

This block of code creates the actual user interface. The `GUI` constructor takes 5 arguments:

1. The `SliderMatrix` to use for the program
2. Minumum x value on the graph
3. Maximum x value on the graph
4. Minumum y value on the graph
5. Maximum y value on the graph

```
GUI gui = new GUI(sm, -2.0, 2.0, -3.0, 10.0);
```

In order to display the functions you have created, use the `addFunction()` method. This takes one argument of type `Function`. Remember to initialize your function first before setting it.

```
gui.addFunction(f1);
gui.addFunction(f2);
```

If you would like to use an icon to label the axes on the graph, use the `setAxesIcons()` method. This method takes the filenames of the two icons you have. The first argument takes the image file for the independent axes and the second is for the dependent axes.

```
gui.setAxesIcons("axis_x.jpg", "axis_y.jpg");
```

The `GUI` class extends from JPanel, so it can be treated as such. To add the GUI to the Applet window, use the `add()` method in the `Applet` class.

```
add(gui);
```

2. Important Classes

2.1. Overview

To sum up the previous section, the constructors for two of the three classes you need to be familiar with are listed below:

```
public SliderMatrix(int capacity);

public GUI(SliderMatrix sm,
  double minx,
  double maxx,
  double miny,
  double maxy);
```

The third, the `Function` class is an abstract class.

2.2. `GUI` and `Graph`

There are a number of methods available to customize your GUI. In order to gain access to the `Graph` object, use the `getGraphObject()` method in `GUI`. The following two `Graph` methods are of particular interest.

```
    /** Set the frequency in which tick marks and labels show up
  * on the x-axis..
     *
     * @param d value between tick marks
  */
 public void setTickIncrementX(double d);

    /** Set the frequency in which tick marks and labels show up
  * on the y-axis..
     *
     * @param d value between tick marks.
  */
 public void setTickIncrementY(double d);
```

Consult the JavaDoc documentation for more public methods available in the `Graph` object.

You may also use the second constructor

```
public GUI(SliderMatrix sm, double minx, double maxx, double miny, double maxy,
  boolean showxzoom,
  boolean showyzoom,
  boolean showdensity,
  boolean showparams,
  boolean showlimits);
```

to set which GUI components are displayed. The only difference between this constructor and the first constructor is the five boolean parameters which determine which GUI componets are displayed.

2.3. `Function`

This is the only class we have not discussed up until now. The code for the functions to be graphed will be the bulk of what a `GrapherGUI` user must write. It is extremely important to understand this class completely. Here is the full source code for this class:

```
package GrapherGUI;
import java.awt.*;

public abstract class Function {
 public abstract double f(double in, SliderMatrix p);

 public Color color() {
  return (new Color(0, 0, 255, 153));
 }

 public abstract String name();
}
```

Note that `Function` is an abstract class. This means that it cannot be instantiated. Instead, in order to create a function to be graphed, we must create our own class that inherits from `Function`. The process of graphing involves the `Graph` object providing a set of parameters (your `SliderMatrix` object) and the value of the independent variable to your `Function` object. `Graph` does this a number of times (the density) to get a set of coordinates to draw. In order for this to work, you must have two methods in your class. Here is an example from `Web.java`:

```
class MyFunc2 extends Function {
    public double f(double in, SliderMatrix p) {
        double g, h;

        try {
            g = p.val("g");
            h = p.val("h");
        } catch (Error e) {
            return 0.0;
        }
        return in * in * in * h / g;
    }

    public Color color() {
        return Color.green;
    }
    public String name() {
        return "x^3 * h / g";
    }

}
```

Note that `public double f(double, SliderMatrix)` calculates the dependent variable value. Note also that `public String name()` returns the string name. This string is used when the program displays the key. Also, `public Color color()` overrides the default color of blue.

So how do we access the parameters? Examine the following lines:

```
try {
 g = p.val("g");
 h = p.val("h");
} catch (Error e) {
 return 0.0;
}
return in * in * in * h / g;
```

The method `val()` in `SliderMatrix` class returns the current value. It takes an identifier string as input. So if you added a parameter with the name `"omega"`, you would call `p.val("omega")` to get its value. The `try try_codeblock catch handle_codeblock` is required because `val()` throws an error if it cannot find the parameter requested. Upon error, the program will execute `handle_codeblock`. In our case, we just return 0.0. This is not recommended. In fact, an error of this type should usually be fatal. Finally, we return the value. This class defined a very simple function: x^3 * (h / g)

We are not limited to just having three predefined methods in a class inheriting from `Function`. We can have as many helper methods as needed, but `f()` and `name()` are required or the program will not compile. For more on abstract classes, consult the Java OOP tutorial (http://java.sun.com/docs/books/tutorial/java/javaOO/abstract.html).

3. Compiling Your Work

3.1. Files

The following is a listing of the source code directory. As you can see, the sources for the for the `GrapherGUI` library are all located in one directory. You should place the compiled class files or a copy of the GrapherGUI directory in the same directory as your custom java sources. Our two files are called `Web.java` and `Standalone.java`. Notice also that by default, all custom image files should be placed in the same directory.‘

```
bash-2.05b$ ls -al *
```

```
-rw-r--r--    1 bshi     users        1906 Jul 10 18:37 Standalone.java
-rw-r--r--    1 bshi     users        1435 Jul 10 15:38 Web.java
-rw-r--r--    1 bshi     users        8726 Jul  4 11:32 axis_x.jpg
-rw-r--r--    1 bshi     users        8850 Jul  4 11:31 axis_y.jpg
-rw-r--r--    1 bshi     users         182 Jul  9 20:01 index.html
-rw-r--r--    1 bshi     users        8804 Jul  4 11:29 param_g.jpg
-rw-r--r--    1 bshi     users        8595 Jul  4 11:30 param_h.jpg

GrapherGUI:
total 53
drwxr-xr-x    2 bshi     users         272 Jul 11 10:12 .
drwxr-xr-x    4 bshi     users         344 Jul 11 10:12 ..
-rw-r--r--    1 bshi     users        1312 Jul  8 11:41 Function.java
-rw-r--r--    1 bshi     users        7938 Jul  9 19:56 GUI.java
-rw-r--r--    1 bshi     users       18670 Jul  9 18:52 Graph.java
-rw-r--r--    1 bshi     users        4590 Jul  9 18:40 ParameterPanel.java
-rw-r--r--    1 bshi     users        3774 Jul  8 12:42 SliderMatrix.java
-rw-r--r--    1 bshi     users         877 Jul  8 14:35 Util.java
```

3.2. Compiling

Using the above directory structure as an example, to compile `Web.java`, **cd** into the same directory and use the following command:

javac Web.java

After compiling the Applet, it would be most convenient if we did not have to lug around the whole directory in order to utilize it. Using JAR (the Java archiver), we can package all the class files and data files (just images in this case) into one file. This way, we can easily distribute copies of the program. So after you have compiled the program using **javac**...

jar Web.jar *.class GrapherGUI/*.class *.jpg

This takes every single class file you have created and the image files you need for your program and packs them into a file called `Web.jar`. Now the only thing left to do is to create an HTML file into which we will embed the Applet.

Place the following lines anywhere in the body of an HTML page. A simple full example can be found here (applet.html.html). (NOTE: you may have to tweak your HEIGHT and WIDTH to get everything to show depending on how large you have set the visual area in `GUI`. 530x720 will work for most people if you do not change the library code. Consult the source documentation for details.)

```
<applet
```

```
 CODE="Web.class"
 ARCHIVE="Web.jar"
 HEIGHT="720"
 WIDTH="530">
</applet>
```

All that needs to be done now is to place your HTML file and JAR file into a directory that is used by your webserver.

3.3. Alternative Organizational Schemes

The example provided in section 1.2 places three classes (the main applet class and two `Function` classes into a single file. This is not the only way to organize your classes.

If the code for your methods are very long, you can place the code for each of your methods into a separate file. If you do this, remember to make them public:

```
public class SpecialFunction extends Function {
   // ... code for this class ...
}
```

If you have a number of functions you wish to reuse, you can organize them into your own personal library of functions. This way, you have quick access to a collection of functions. To do this, create a directory called what you want the package name to be. Place all your function source files in that directory. If your package is called `functionlibrary`, then you would add the following line to the beginning of each file in the package directory

```
import functionlibrary.*;
```

to gain access to the functions included in the library. For more information, consult the Sun documentation on creating packages (http://java.sun.com/docs/books/tutorial/java/interpack/createpkgs.html)

3.4. Packaging

There are a number of ways you can archive everything. In order to run a GrapherGUI program or applet, you will need all the GrapherGUI class files and the class files for Function and Applet/Application classes that you have created.

One option is to not use anything packaging system at all. This is not recommended unless you are just testing because **javac** generates many class files which all need to be up to date in order for the program to run. If you are copying the package and miss one file, the application will fail.

For more information, consult the Sun documentation on JAR (http://java.sun.com/docs/books/tutorial/jar/basics/index.html).

4. Advanced Topics

4.1. Code Documentation

In addition to this tutorial, there is a folder of documentation on the source code for the library. Perhaps you do not need certain buttons or GUI components currently available or perhaps you wish to add a button. Consult the JavaDoc documentation (../index.html) on GrapherGUI for an introduction on the source.

4.2. Creating a Standalone Application

Because all the work is done within the GUI class, there is quite a lot of freedom in terms of where you place the graphing utility. We have already gone through an example of how to create an applet containing a grapher. This section describes how to create a standalone application using GrapherGUI. The full source code can be found here (Standalone.java.html).

When creating an application, you must declare the following in your main class. The first thing to do is to create your `Function` objects, `SliderMatrix` object, and `GUI` object. After that, just cut and paste the rest because the rest of the code will remain consistent across any program. If you are curious what the code does, consult Sun's documentation on JFrame (http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JFrame.html).

```
static public void main(String[] args) {

 // ... SliderMatrix, Function, and GUI setup here ...

 JPanel p = new JPanel();

 p.add(gui);

 JFrame f = new JFrame("Grapher");

 f.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent we) {
   System.exit(0);
  }
 });
 f.setSize(510, 720);
 f.setLocation(300, 200);
 f.getContentPane().add(p);
 f.setVisible(true);
 f.show();
}
```

To run an application from the command line, use the following command "**java Standalone**" If your class name is Standalone. Like the first example, you can name your class whatever you want. This procedure is useful for testing as it is not necessary to make an HTML file or to load a browser every time you wish to test your program.

Because `GUI` inherits from the JPanel (http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JPanel.html). class, you could even embed whatever grapher you create inside a larger Java program as an internal utility. That is outside the scope of this document but this is in practice not difficult.

4.3. Fortran/C/C++

Java is regarded as being too slow for mathematically intensive tasks. One way of optimizing the speed of your application is to use C or Fortran for mathematical routines. This is possible through the use of the Java Native Interface (JNI (http://java.sun.com/docs/books/tutorial/native1.1/)).

We will run through an example. The three new files to study are

- `NativeFunction.java` (NativeFunction.java.html)
- `NativeFunction.h` (NativeFunction.h.html)
- `NativeFunction.c` (NativeFunction.c.html)

4.3.1. The Java Portion

We want a Function that uses natively compiled code. An example of such an interface between Java and C is given in `NativeFunction.java`. This class is very similar to the function classes we defined in earlier examples, most importantly in that it extends the `Function` class.

The method `f()` and `name()` are defined but unlike our other functions, we do not do any calculations. `f()` grabs the values of two parameters and passes them to a new method called `native_f`.

```
public native String name();
public native double native_f(double in, double p1, double p2);
```

```
static {
 System.loadLibrary("NativeFunction");
}
```

The `native` keyword indicates that the function will be implemented natively. At the end of this code snippet, we load a library called "NativeFunction". This means that the program will try to look for the native functions defined in this class (`name()` and `native_f`) in a shared library called `libNativeFunction.so` (Note: on Linux/Unix systems)

4.3.2. The C portion

Now that we have an interface between Java and C, we need to write the native code. How do we know what to functions to define? After compiling `NativeFunction.java`, use the `javah` tool to create a C header file for your native code. Run `javah`:

```
$ javah NativeFunction
```

Note that `javah` takes a class name and not a filename for an argument. We get the following to prototyped functions:

```
/*
 * Class:     NativeFunction
 * Method:    name
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_NativeFunction_name
  (JNIEnv *, jobject);

/*
 * Class:     NativeFunction
 * Method:    native_f
 * Signature: (DDD)D
 */
JNIEXPORT jdouble JNICALL Java_NativeFunction_native_1f
  (JNIEnv *, jobject, jdouble, jdouble, jdouble);
```

The first defines the function which will be used for `NativeFunction.name()` and the second for `NativeFunction.native_f()`.

4.3.3. The Code

At first the huge obfuscated variable names and types might seem daunting, but don’t worry -- we won’t really need to use many of them.

Each JNI function has at least two arguments. The following arguments are the arguments defined in NativeFunction.java. Our `native_f()` function is very simple - we want to plot x^2 * (g - h). Note that the native implementation of this (`Java_NativeFunction_native_1f`) function returns a `jdouble` and takes three other `jdouble` arguments in addition to the two that are always there. Remember back to our Java source; the first is the independent variable, the others are parametes passed in by our Java program.

You can do a direct cast from `jdouble` to `double`:

```
return (jdouble) ((double) in * (double) in * ((double) g - (double) h));
```

This simplifies matters a bit. Finally, once you have your dependent variable, just cast it back into a `jdouble` and return it. Our `name()` function (`JNIEXPORT jstring JNICALL Java_NativeFunction_name`) is very straightforward. All we need to do is return a Java string by using the `NewStringUTF()` method that is part of our `env` variable:

```
return (*env)->NewStringUTF(env, "My Function Name");
```

This function should never get more complicated than one line. In fact, you probably don't need to implement a native version of this function, but we do so here to show that it is possible to manipulate more than primitive variable types within your native code.

To compile a shared library using GCC under Linux/Unix, issue the following commands:

```
$ gcc -c NativeFunction.c -o NativeFunction.o
$ gcc -shared -o libNativeFunction.so NativeFunction.o
```

Make sure that the compiler knows where to find `jni.h` and the Java development libraries. You can use the -I and -L arguments for GCC to set this. Now place `libNativeFunction.so` in a directory where the linker can find it. Alternatively, you can set the $LD_LIBRARY_PATH to point to the directory the shared library is located and run whatever Java application uses it.

For more complex functions, you should probably create a separate function(s)/file(s) and call them.