The graph library: simple graphical steering for scientific applications

Alexander Wagner

April 23, 2003

Contents

1 History

In my first year as a physics student at Bielefeld University in 1992, I met Johannes Schlosser. What began as a study group for physics problems soon developed into a great friendship and an intense and fruitful collaboration on exploring what computers could do.

At this time a computer would be in "text mode" as standard and only for special problems would a "graphics mode" be activated. At this time only Apple computers broke with this paradigm and Microsoft was only a small company making the Microsoft DOS as well as a few text programs. A graphics resolution of 600:300 was a great resolution and color was a luxury we could not afford. We each bought an 8068 IBM compatible computer and began checking out what we could do with it. Switching the monochrome Hercules graphics card into graphics mode was a major challenge. We started using Turbo Pascal as our programming language of choice with a few Assembler inline additions.

But seeing the graphics on the screen is rather fleeting, so we also wanted to be able to print it. There was the option of a screen print, but that meant limiting the print output to the low resolution of the screen, even though I had invested in an NEC P6c, a color dot printer. So we wanted to make sure that we could print out our graphics with maximal resolution and in color on our printer. But how can you transfer a picture from the screen to the printer and increase the resolution? The solution we found was to have two sets of basic graphic routines: one would write to the screen, the other would write to a set of memory which was a bitmap of what we would then write to the printer. We realized that if you wrote your program so that all sizes would be relative to the X and Y dimensions of your medium, they would look right, no matter if they would be printed on a screen or printed to the printer.

On this level, we managed to program a graphical user interface which was supposed to be easy to adapt for new programs. If you wanted to print something you could use the same routines that you had used to create your graphics on the screen. It came as a total revelation to us when, a few years later, Peter Seroka introduced us to the X windows system common on Unix workstations. Being able to open windows on other computers seemed like a little miracle to us. A little earlier Linus Torwald had developed the Linux kernel and now the first working versions of the Linux operating systems became available and they were free! Another revelation there. So with the help of Linux (version 0.92, I believe) we could transform our computers (80286 IBM compatibles at this time) into Unix workstations.

This signaled the change from Pascal to C (which only our friend Tim Scheideler was using at the time) and X windows for our graphics needs. Previously all printing was different for all printers, but the people of Adobe Inc. had developed PostScript, a portable graphics language. And with the help of ghostscript you could (and still can) print any postscript file on any printer. So we decided to now use PostScript files as the output which could then be printed *everywhere*.

At this time both Johannes and myself were working on our Diplomarbeit (masters thesis) at Bielefeld University. Not surprisingly, we were both doing computational projects and were heavily relying on graphical output. At this point we had actually developed two separate graphics libraries. Johannes had true three-dimensional capabilities whereas I had developed basic menus, the printer support, and coordinate systems. Because Johannes was the much more organized of the two of us we used the structure of this graphics library and added my extensions into his code to create what remains to this day the basic foundation of the library: a set of basic two dimensional graphics routines which interface either xlib of the X windows system or PostScript routines, a library for the representation of three dimensional scenarios. This was supported by a set of menu functions that dealt with mouse interactions.

After we successfully used the graphics libraries for our Diplomarbeit Johannes went on to a rapidly advancing career at IBM Deutschland whereas I went on to graduate school at Oxford. For my graduate work I picked a numerical project again, but this time I dealt with complex fluids which needed a number of different representations that our graphics library did not yet support. I included support for density plots and contour lines, but whenever I wrote a new program I still copied lots of old code from the last programs to reuse them in the new programs. And then I realized that anytime I copied code from other programs, it meant that it should be part of the library and not part of the user code. It took quite a few numbers of iterations to reduce the amount of code the user had to write to use the graphics code.

In 1998, at the end of my D.Phil. (Ph.D.), I considered briefly founding a company for simulating complex fluids and using the graphics as a part of this commercial enterprise. But I soon realized that my heart was with science and I did not enjoy the prospects of meeting with venture capitalists to fund my software company (and that was at the beginning of the Internet bubble). Instead, I continued my academic career with postdoctoral positions at MIT and Edinburgh University. At each of these places I realized that scientists as a whole don't tent to have easy access to the data in their computer simulations and that the approach I had taken was quite unique.

Now I have a faculty position at North Dakota State University and I want to give you the ability to steer your computer simulations and to monitor them closely. Considering how much Johannes and I have benefited from the free software project we have decided to release our library under the GNU public license to make sure that it is available for free to anyone.

2 So what does it do?

The project began as a library for real time three-dimensional visualization. We had video games in mind, as well as scientific visualization. In the end we never finished the video game, but the scientific visualization part became more and more important. From these beginnings a a library developed that is supposed to make scientific visualization as easy as possible. In this introduction I will leave out the details of the graphics machine and focus only on the use of library functions for scientific visualization.

I hope that a full documentation of the software will become available in the future to make it easier for others to extend the library.

3 Scientific Visualization

Imagine that you want to simulate a fluid mixture that is phase-separating. For scientific reasons you may be interested in some statistical properties of this process, but how can you be sure that the program is really doing what you expect it to do? And what do you do if you want to find out what the effect of different parameters is? There will often be quite a few different parameters that you need to control.

My vision for this library is that it will help scientists to transform their simulations which are performing their instruction in a way that is hard to observe, into a transparent object where you can look at all times at the relevant variables and where you can change the parameters to see what will happen. This changes the nature of computational research: suddenly you are able to see and manipulate the system you simulate directly. You will now be able to see what is going on in the system, and if something does behave differently than you expect you have probably found a bug in your program and you can eliminate it faster. And if it isn't a bug you have just learned something interesting about the system. And once in a while you will be the first to notice and you have made a discovery that you might not have made without being able to observe the system so closely (this is what happened to me a lot of the time).

This is not to suggest that you give up the more traditional approach of defining curves for which you can make analytical predictions or which you can compare to experiments. But I feel that it is a very valuable extension of this approach.

4 Examples

Say you want to write a code that solves the one-dimensional diffusion equation with periodic boundary conditions. Don't worry if you don't understand this particular problem. It is just a specific example of a computational problem that you might have. Understanding the details of this is not important in order to follow the main points of this section. The diffusion equation is given by

$$\frac{\partial \rho}{\partial t} = \kappa \frac{\partial^2 \rho}{(\partial x)^2} \tag{1}$$

You can model this equation by writing a small lattice Boltzmann program. You could use many other ways, but lattice Boltzmann is close to my heart. If you are interested in the details of this algorithm you can find an explanation in appendix B. Your program[1] might look something like this:

```
#include <stdio.h>
#include <math.h>

#define size 100
static Repeat=1000;
static double f0[size],f1[size],f2[size], omega=1, T=0.3;
```

[1]This program is included as `prog1.c` in this distribution.

```
void init(){
  int i;
  double density;
  for (i=0;i<size;i++){
    density=(2+sin(2*M_PI*i/size));
    f0[i]=density*(1-T);
    f1[i]=density*T*0.5;
    f2[i]=density*T*0.5;
  }
}

void iterate(){
  int i;
  double density,tmp1,tmp2;
  for (i=0;i<size;i++){
    density=f0[i]+f1[i]+f2[i];
    f0[i]+=omega*(density*(1-T)-f0[i]);
    f1[i]+=omega*(density*T*0.5-f1[i]);
    f2[i]+=omega*(density*T*0.5-f2[i]);
  }
  tmp1=f1[size-1];
  tmp2=f2[0];
  for (i=1;i<size;i++){
    f1[size-i]=f1[size-1-i];
    f2[i-1]=f2[i];
  }
  f1[0]=tmp1;
  f2[size-1]=tmp2;
}

int main(){
  int i;
  init();
  for (i=0;i<Repeat;i++) iterate();
  return 0;
}
```

This is a very simple program which currently only consists of the main computational kernel. It does not “do” anything yet, since it has no output. It has been my general experience that the actual computation can usually be written in a succinct bit of code whereas the analysis and output make up the main part of the code. And this analysis tends to be so individual that there is little guidance in how to write this bit of the code.

Now we want to see what the code actually does. So we would like to see what the density is at each new time step and how it evolves. We might want to be able to re-initialize the simulation and maybe to initialize it with different initial conditions. Also, we might want to be able to change the size of the simulation. And on the fly, we might want to be able to change the diffusion constant which in this simulation is $\kappa = (1/\omega - 0.5)T$. So we might want to change the two parameters independently so that we can examine how the numerical errors differ for the two approaches.

This may sound like quite a laborious task, but this is exactly what the mygraph library was designed to do. What we have to do is to tell the library which data we want to look at, give it some hints about its size, and it will be able to display it appropriately. We will also need a few other control variables to be able to tell the program that it should pause, run for only one step at a time, or that it should re-initialize. So this program, with a full graphical user interface (GUI),[2] would look like this:

```
#include <stdio.h>
#include <math.h>
#include <mygraph.h>

#define SIZE 100
static int size=SIZE,Repeat=1000,done=0,sstep=0,pause=1;
static double f0[SIZE],f1[SIZE],f2[SIZE], omega=1, T=0.3,Amplitude=1;
static double density[SIZE];
static int densityreq=0;


void init(){
  int i;
  for (i=0;i<size;i++){
    density[i]=(2+Amplitude*sin(2*M_PI*i/size));
    f0[i]=density[i]*(1-T);
    f1[i]=density[i]*T*0.5;
    f2[i]=density[i]*T*0.5;
  }
}

void init2(){
  int i;
  for (i=0;i<size;i++){
    if (2*i>=size) density[i]=2+Amplitude; else density[i]=2-Amplitude;
    f0[i]=density[i]*(1-T);
    f1[i]=density[i]*T*0.5;
    f2[i]=density[i]*T*0.5;
  }
}

void iterate(){
  int i;
  double tmp1,tmp2;
  for (i=0;i<size;i++){
```

[2]This program is distributed as `prog2.c` with this distribution.

```
    density[i]=f0[i]+f1[i]+f2[i];
    f0[i]+=omega*(density[i]*(1-T)-f0[i]);
    f1[i]+=omega*(density[i]*T*0.5-f1[i]);
    f2[i]+=omega*(density[i]*T*0.5-f2[i]);
  }
  tmp1=f1[size-1];
  tmp2=f2[0];
  for (i=1;i<size;i++){
    f1[size-i]=f1[size-i-1];
    f2[i-1]=f2[i];
  }
  f1[0]=tmp1;
  f2[size-1]=tmp2;
}


void GUI(){
  DefineGraphN_R("Density",density,&size,&densityreq);
  StartMenu("GUI",1);
    DefineDouble("T",&T);
    DefineDouble("omega",&omega);
    StartMenu("Restart",0);
      DefineMod("size",&size,SIZE+1);
      DefineDouble("Amplitude",&Amplitude);
      DefineFunction("Restart sin",&init);
      DefineFunction("Restart step",&init2);
    EndMenu();
    DefineGraph(curve2d_,"Density graph");
    DefineBool("Pause",&pause);
    DefineBool("Single step",&sstep);
    DefineInt("Repeat",&Repeat);    DefineBool("Done",&done);
  EndMenu();
}

int main(){
  int i;
  init();
  GUI();
  while (!done){
    Events(1); /* Whenever there are new data the argument of
  Events() should be nonzero. This will set the
  requests for data so that you can calculate them
  on demand only. For this simple program you can
  always set it to one. */
    DrawGraphs();
    if (!pause || sstep){
      sstep=0;
      for (i=0;i<Repeat;i++) iterate();
    } else {
      sleep(1);/*when the program is waiting it returns the
 CPU time to the operating system */
    }
  }
  return 0;
}
```

Let us go briefly through the new additions to the program which implement the GUI. The function `GUI()` tells the graphical user interface about the data to display and the variables we want to be able to change interactively. Firstly we have the one-dimensional density field that we want to display. This data gives for a section of the natural numbers `N` one real number `R`. The function that tells the GUI about this is

```
  DefineGraphN_R("Density",density,&size,&densityreq);
```

The first argument is a string which corresponds to the name of the data as it will appear in the menus. The second argument is the data you want to display. More exactly, it is a pointer to the first element in the array. The third argument is a pointer to a variable giving the size of the data. We use a pointer rather than simply a number because if the size of the data changes during the simulation (and this change is reflected in a change of the variable `size`) the GUI will know about this and display the data correctly. Also note that we made the density an array, not just a temporary variable of the iterate routine, so that we can display it.

Then we start the menu for the GUI.

```
  StartMenu("GUI",1);
```

The first argument is the name of the menu, and the second argument is either 0 or 1. If it is 1 this indicates that the menu will be initially displayed. Since this is the first menu, it should certainly be displayed. Next we define two `double` variables as menu items

```
    DefineDouble("T",&T);
    DefineDouble("omega",&omega);
```

These functions have two arguments: the name as it appears in the menu and the address of the variable. We now define a sub-menu with a new `StartMenu()` function as above.

```
    StartMenu("Restart",0);
      DefineMod("size",&size,SIZE+1);
      DefineDouble("Amplitude",&Amplitude);
      DefineFunction("Restart sin",&init);
      DefineFunction("Restart step",&init2);
    EndMenu();
```

There are three new routines:

`DefineMod("size",&size,SIZE+1)` inserts a menu item for the variable size and only allows it to vary from `0` to `SIZE`.

`DefineFunction()` allows you to start functions which don't have any arguments to be called at the press of a button. The first argument of this function is again the name as it appears in the menu and the second is the address of the function you want to be able to call. Note that we introduced a second initialization routine that initializes the density as a step function to make things a bit more interesting.

`EndMenu()` tells the GUI that this is the end of the sub-menu.

Next we have a special menu item to display line graphs.

```
    DefineGraph(curve2d_,"Density");
```

The routine that does this is `DefineGraph()`. The first argument of this function is an integer that is represented by the name `curve2d_`.[3] The rest of the GUI implementation should now be self-explanatory.

```
    DefineBool("Pause",&pause);
    DefineBool("Single step",&sstep);
    DefineInt("Repeat",&Repeat);
    DefineBool("Done",&done);
  EndMenu();
```

After we have successfully initialized the GUI there are two more library functions that you have to be aware of. Firstly

```
    Events(1);
```

For any program with a graphical user interface you have to give the library functions a chance to react to mouse and keyboard events. It is this routine that does it. So you have to make sure that you call `Events()`[4] regularly so that windows can be redrawn, resized and mouse clicks can be acted upon. The second routine is

```
    DrawGraphs();
```

This routine will draw all the graphs that have need to be displayed.[5] One other consideration to keep in mind is that interacting with a user is slow. So you really don't

[3]There are several other types corresponding to two-dimensional density and vector plots, three-dimensional graph representations, three-dimensional contour plots, etc. This is discussed in section ??.

[4]Whenever there are new data the argument of `Events()` should be nonzero. This will set the requests for data so that you can calculate them on demand only. For this simple program you can always set it to one.

[5]But why is this not simply part of the `Events()` function? The reason lies in the following consideration: often there are quantities that you want to monitor, but that are not actually required for the computation. Having two routines allows you to calculate only when they are needed. This is the reason that each data is associated with a request flag. If the user chose to observe this data, the request flag would be set in the `Events()` routine and you could write a routine that would calculate this data before you called the `DrawGraphs();` routine.

want to call the `Events()` routine too often because that would make your program slow.

Now I just want to make a few more remarks about some useful steering parameters that I tend to use for simulations. We want to be able to stop the calculations temporarily to look at the data in leisure, to be able to step through the calculation one iteration at the time and to re-initialize the calculation. And we want to be able to quit the computation at the press of a button. So we define the variables

variable	description
`done`	this is equal to 0 until the simulation is finished
`pause`	this is equal to 0 unless the simulation is paused
`sstep`	this variable gets set to one to run the simulation for a single step
`Repeat`	this variable is set to the number of steps that the simulation should be run until the `Events()` function is called again

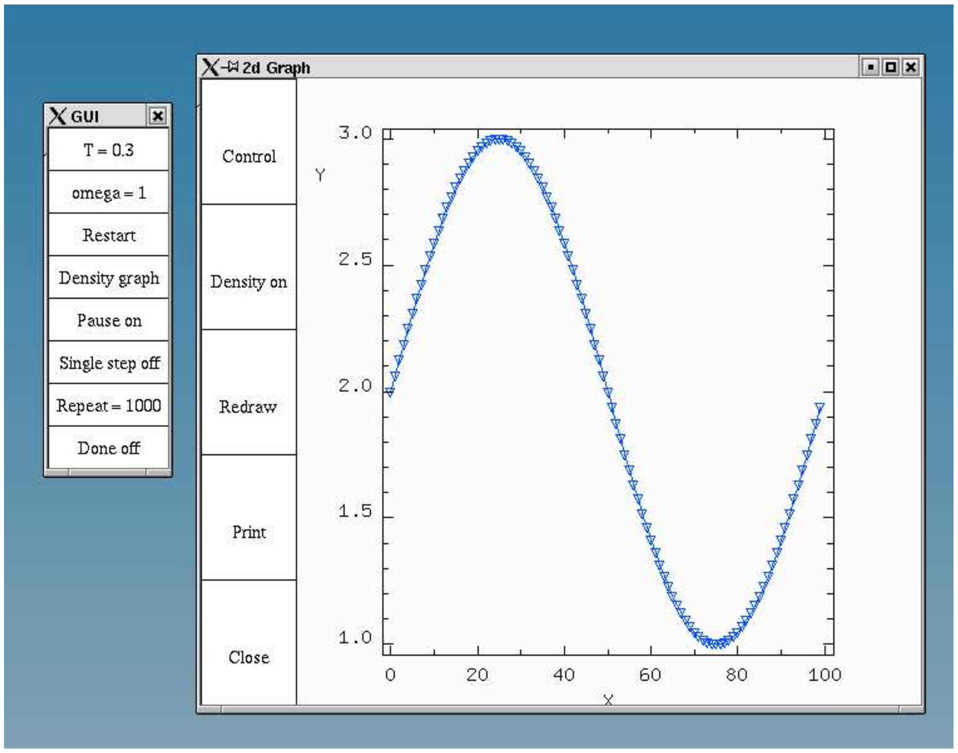
With this explanation of the variables the logic of the main routine should be obvious.

```
while (!done){
  Events(1);
  DrawGraphs();
  if (!pause || sstep){
    sstep=0;
    for (i=0;i<Repeat;i++) iterate();
  } else {
    sleep(1);/*when the program is waiting it returns the
                CPU time to the operating system */
  }
}
```

The only other comment I want to make is regarding the `sleep(1)` statement. This routine takes control away from the program and gives it to the operating system. Why would you want to do this? This is mainly a question of consideration for others and for any of your own programs that might be running at the same time. Without this statement the program would always use all the CPU cycles it can get its hands on, even if it is doing nothing else than waiting for any input you might want to give it. And while it is waiting, it might as well give some time to programs that really need it.

4.1 Compiling the program

In order to compile the program you have to tell it where to find the header files for the simulation and where to find the library. Then you need to link your program to the graph and X11 libraries. Assuming that you use the standard implementation (see A Installing the library), the command will look something like



T = 0.3
omega = 1
Repeat = 1000

Figure 1: Screen-shot of the GUI after a graphics window has been opened and the density graph has been selected. If you right-click on the density option you can choose symbols and colors for the graph.

```
cc -I ~/include prog2.c -L ~/lib -L /usr/X11R6/lib -lm -lgraph
-lX11 -o anim2.out
```

This assumes that you have installed the library in your home directory. For system-wide installations you would not need the special `-I  /include` and `-L  /lib` options. For more explanations see the man pages of your c compiler.

4.2 Running the program

If you now type

```
./prog2
```

you will see the GUI appear. The GUI is shown in Figure 1. First you may want to click with the left mouse button on the "Density graph" button to see the density. Then in the density graph you may want to select the "Density" with the left mouse button. If you click with the right mouse button on the "Density" field you can choose several properties of the graph. If you want to remove the new menu simply right-click the "Density" button as well. This is a general method to remove menus that you have created. The only exceptions are graphics windows because you can create an unlimited number of them.

Now clicking on the “Control” button will give you a menu that allows you to choose several properties of the graph. You may want to toggle the “AutoScaling” button. Next right-click on the “Repeat = 1000” button in the main menu. A cursor will appear and you can change this value to 1. Make sure that you finish the input by pressing enter. The program will not allow you to do anything else before you do this.

Now, if you are ready to look at the evolution of the density, press the “Pause” button and watch the evolution of the density. You can see that the initial density fluctuation decays and leaves us with a homogeneous solution. We actually know that analytically the sin function with an exponentially decaying amplitude is a solution of the diffusion equation. So now press “AutoScaling” to on and press the Restart button. In the new menu press “Restart now”. Now you can see the evolution of the density and the form indeed remains constant (except for the moving scale) so that we see that our numerical approach is in agreement with the analytical solution.

To finish the program simply press the “done” button.

5 Reference section

In this section you can find the list of all the user commands (and some functions that you are less likely to need) that you are going to need to visualize and steer you programs.

6 Does it work with other programming languages?

You can use this library on any system supporting the X windows system and able to link c-libraries. I have used it successfully within the parallel programming language ZPL.

A Installing the library

First you need to obtain the `graph.tar.gz` file either from the web at http: www.physics.ndsu.nodak.edu/people/awagner/graph.html or from another source and put it in your home directory. You then type

```
tar xzvf graph.tar.gz
cd c/graph
make install
```

If you don’t get error messages this will successfully install the library in your system.

B The lattice Boltzmann algorithm

This clearly needs to be written, but it should probably be a different document on lattice Boltzmann methods. It might be nice to link these on the web though.

function	description
`DefineGraphN_R(char *Name,double *gd,int *dim,int *req)`	Simple one dimensional arrays
`DefineGraphN_RxR(char *Name,double *gd,int *dim,int *req)`	One dimensional arrays of points (x,y).
`DefineGraphN_RxRp(char *Name,double **gd,int *dim,int *req)`	Pointer to an one dimensional array of points. Needed if the size of the data field will be resized with `realloc()`.
`DefineGraphNxN_R(char *Name,double *gd, int *dim1,int *dim2,int *req)`	Two dimensional array.
`DefineGraphNxN_RxR(char *Name,double *gd, int *dim1,int *dim2,int *req)`	Two dimensional array of vectors.
`DefineGraphNxN_RxRxRt(char *Name,double *gd, int *dim1,int *dim2,int *req)`	Three dimensional field. *Still waiting for the implementation of the 3 dimensional contour.*
`DefineGraphContour3d(char *Name,Contour **gd, int *dim1,int *dim2,int *dim3,int *req)`	A special data type to define contour, especially from a parallel program.
`SetDefaultColor(int c)`	Default color for graphs defined after this command. See Table 2.
`SetDefaultLineType(int s)`	Default line type is 1 (solid line). 0 means no line.
`SetDefaultShape(int s)`	Default shape according to table 2.
`SetDefaultSize(double s)`	Default size is one.
`SetDefaultFill(int f)`	Default fill is off (0).
`NewGraph()`	Start a new set of graphs.
`SetActiveGraph(int)`	Set this before you call `DefineGraph()`.

Table 1: All commands used to register data that you want to visualize

Name	value
NoShape	0
UpTriangle	1
DownTriangle	2
Square	3
Circle	4

Color functions	English
schwarz()	black
weiss()	white
rot()	red
gruen()	green
blau()	blue
gelb()	yellow

Table 2: Shapes and standard colors. *Colors will be translated soon.*

Function	description
`StartMenu(char *Name,int installed)`	Start a new menu with name *Name*. It will be visible as you start the program if *installed* is 1
`StartBoolMenu(char *Name,int *b)`	This will also start a menu, but the menu is also a Boolean field (see below) with name *Name* and address **b*. To change the value of *b* left click the button, to open the menu right click the button.
`EndMenu()`	Each menu has to be terminated by an `EndMenu()` command.
`DefineInt(char *Name,int *myint)`	Menu item of type int
`DefineLong(char *Name,long *myint)`	Menu item of type long
`DefineMod(char *Name,int *myint,int mod)`	Menu item of type int. The value is constraint to be between `0` and `mod-1`
`DefineBool(char *Name,int *myint)`	Integer that can only take the values 0 and 1. In the menu this is represented as “off” and “on” respectively.
`DefineDouble(char *Name,double *mydouble)`	Menu item of type double
`DefineFloat(char *Name,float *myfloat)`	Menu item of type float
`DefineString(char *Name,char *s,int maxlen)`	Menu item for a string with maximum length maxlen. *There is a bug that still allows you to write strings longer than maxlen.*
`DefineFunction(char *Name,fp myfp)`	This function allows you to call a function. This function may not have an argument.
`DefineItem(int vartype)`	This is a function to add special items like the print button (`print_` and `dopint_`), a close button (`close_`) or a Redraw button (`newdraw_`). You will probably not need to use these except maybe the `close_` option.
`DefineGraph(int vartype, char *Name)`	Adding buttons for different kinds of graphics with arguments given by table 5.

Table 3: Menu commands

Type	numerical value	description
`curve2d_`	11	displays `N_R`,`N_RxR`, and `N_RxRp` data types as two dimensional curves.
`contour2d_`	9	displays `NxN_R` data types as density and contour plots and `NxN_RxR` data types as vector plots.
`graph2d_`	8	displays `NxN_R` data types as three dimensional graphs.
`contour3d_`	10	displays `NxNxN_R` data types as three dimensional contour plots. *Not yet fully implemented.*

Table 4: Different options for the `DefineGraph()` routine.