Python Programming in OpenGL

A Graphical Approach to Programming

Stan Blank, Ph.D.
Wayne City High School
Wayne City, Illinois
62895

April 8, 2008

Table of Contents

Python Programming in OpenGL/GLUT

Chapter 1 Introduction

Before we begin our journey with Python and OpenGL, we first need to go back in time. History serves many purposes, but one of its more important functions is to provide us with a reference point so that we may see how far we've traveled. We'll go back to about 1980 and the first computer programming class in our high school. We were the proud "owners" of a single new Commodore VIC-20 and an old black and white TV that served as a monitor (almost). There were about 5 or 6 students in the class and we began to learn to program in BASIC.[1] There were no graphics worth mentioning and the only thing I remember is that we made such a fuss about getting the VIC to find the prime numbers from 2 to 997. If memory serves, it took about 30 minutes for the VIC to run this "sophisticated"[2] prime finding program. We had no disk storage and the memory in the computer was 4K.[3] I think the processor speed was about 1 Mhz and might have been much lower[4], but we didn't care because we were computing!

The next step occurred the following year when we purchased 10 TI 99/4a computers for $50 each.[5] They were not much better than the VIC-20, but we at least were able to store programs using cassette tape recorders. Cassette storage wasn't much fun, extremely slow, and unreliable. I remember some slow, crude rudimentary graphics, but nothing that stands out in my mind. Finally, in 1982, things began to get exciting. We were able to purchase several Apple II+ computers with disk drives. We thought we were in heaven! The Apples were neat looking, nearly indestructible[6], and much faster than anything we had used previously. Plus, they could actually produce usable GRAPHICS. Not just crude blocky stuff (which you could choose if you wanted… but why?), but nice points and lines on the screen! These Apples had 64K of memory (all you could ever use… or so we thought) and the disk storage was amazing. We could store 140K of programs on one floppy disk![7] Our prime number generator took only 53 seconds on the Apple, which was over 30 times faster than the VIC- 20. Had I been acquainted with my friend George Francis at that time, we would have been able to do even more with these dependable machines.[8]

Our final conversion was to the PC platform in 1987-88. We now had a lab of 12 true-blue IBM PC's with color monitors and hard drives running Windows 3.0 (or was it

[1] BASIC is a computer language… Beginner's All-Purpose Symbolic Instruction Code. It has been much maligned over the years; unjustly in my opinion.
[2] Here, "sophisticated" means 'brute strength and ignorance". But the program worked and we were thrilled!
[3] This is 4 thousand bytes of memory. Compare this to my current laptop which has 2 BILLION (gigabytes) of memory.
[4] Again, my current laptop has a processor that runs at 2 Ghz, over 2000x faster!
[5] These were truly awful computers. Texas Instruments introduced them at a price of over $1000 and ended up selling them at Wal-Mart for $49.95. I'm not certain they were worth that much.
[6] I personally saw one dropped on a concrete sidewalk. It bounced once or twice and worked fine for several years afterward. No, I wasn't the one who dropped it.
[7] Again, my trusty laptop has a 60 gigabyte hard drive. That's 60 billion bytes. I also have a portable USB "diskless" drive that holds nearly 2000x the capacity of that Apple disk!
[8] UIUC Math Prof. George K. Francis had a lab of Apples then that did some amazing graphics with a 1983 Forth compiler written by one of his colleagues. It would have been nice to have that!

3.1?). By today's standards, they were painfully slow, but at the time we thought that we were cutting edge. Memory was now in the megabyte range and processor speed was over 10 Mhz. I remember plotting Mandelbrot sets in less than 10 minutes, which was relatively fast for that era. We have steadily improved to our present lab setup of PC machines running nearly at 1 Ghz (or faster) with at least 128 mb of RAM (or more) and dedicated video cards for graphics.[9] The computers in our labs are supercomputers compared to where we started! In fact, if we were to take the computer in front of you back to 1980, it would have been one of the fastest on the planet!

So this was a brief history of our high school computer lab. The programming class curriculum followed the lab, as you might guess. We would spend 3 quarters learning to program and then the 4th quarter was reserved for student projects. Invariably, once graphic capabilities were available, all 4th quarter projects would involve graphics. The first graphics on the Apple were slow and rather crude by present standards. They were barely interactive, if at all. Even on our first PC's it would take several minutes to display minimal fractal images. Not so today. With the computers we have in our lab we can create sophisticated graphics that can be manipulated in real-time… something we didn't even dream of back in 1980! It only took me 20 years to realize that my students were trying to tell me something! For the past 5 years we have concentrated on learning computer programming through computer graphics and that is what you will be doing this year. Learning how to program is hard work, but at the same time, it is very rewarding and can be great fun!

So, if a picture is worth a thousand words, how much more valuable is a changeable, interactive, creative graphics scene? You can graph mathematical functions in 2D, 3D, and even 4D. You can create true stereo images! You can design programs to simulate real-world events and you can manipulate the laws of physics and create your own worlds. If you are an artist, the computer becomes your easel. If you like games, you can program your own according to your own specifications. The computer can become a window into your mind and your limitations are governed by your imagination. What you can envision, you can create! Now how cool is that?

Oh, I forgot to say that you can make a fantastic living doing this stuff… just ask the folks at PIXAR.[10]

[9] Previous computers used the cpu and onboard memory for graphics. This made them slow. A dedicated graphics board handles most of the work and has its own much speedier memory. This allows us to create some rather fancy graphics. By comparison, my laptop has 256 mb of video memory alone… more than the system memory of many computers.
[10] You know, the people who made "The Incredibles" and other such movies.

Chapter 2 Needs, Expectations, and Justifications

Section 2.1 What preparation do you need?

In order to successfully complete this class, you will need to have some knowledge of computer operations as a prerequisite. This class is NOT an introduction to computers or a computer concepts class. This is a programming class and I expect that you already know something about computers. You should already be able to use a computer to do tasks such as word processing, gaming, and internet searches. You should know how to install programs and you should know how to follow directions. You don't need to know how to program, although it's certainly OK if you have some programming experience.

In terms of knowledge, you should have some familiarity with algebra and geometry. You don't have to be an "A" student, but you should be comfortable with the concept of variables, equations, points, lines, angles, and coordinates in the x, y, and z axes. Also, you should have some knowledge of physics and physical science, particularly the equations and concepts for location, distance, velocity, and acceleration. If you are uncomfortable at this point, take a deep breath, relax, and simply be prepared to work a bit harder and I think you'll be fine.

Section 2.2 What hardware and software do you need?

If you are physically in my computer science class, then your computer and software needs are already met by our lab computers. If you are reading this on your own, then I should take some time to tell you what minimum hardware and software you will need to complete the examples in the text. At a minimum, you should have a computer with at least 128 mb of RAM and a dedicated video card with at least 32 mb of video RAM. Most current minimum computer systems should suffice.[1] If your computer doesn't quite meet the minimum requirements, then give the programs a try anyway. They'll probably run fine, albeit more slowly.

As far as software is concerned, we will be using a programming language called Python with the PyOpenGL module.[2] We will also use a programming editor called DrPython, although other editors such as SciTE are available. Again, if you are in my class, these programs are already on the lab computers. If you are studying on your own, the programs may be downloaded from the internet at no cost.[3] I use both Windows and Linux without a real preference, although it is probably easier to use the Windows platform in terms of the necessary software installation. All the program

[1] More is better!
[2] You will need to have OpenGL and GLUT installed on your computer. Windows machines already have OpenGL installed. You may need to search online for GLUT and install the glut32.dll file in your system directory.
[3] Use your search engine. DrPython needs another program called wxWindows, also freely available. If you need help with the installations, seek advice from your teacher or friendly neighborhood computer geek. These programs are "Open Source", meaning that they are free.

examples in this text will work on either platform as long as the proper software is installed.[4]

Section 2.3 My Expectations

My expectations are simple. I expect you to make an honest effort to learn the material. If you feel that you are deficient in an area, then take some time and strengthen your knowledge. I expect you to try the programs in this text, to modify the programs, and to create some programs of your own. The exercises at the end of each section or chapter will help achieve fluency in programming and should be attempted with honest effort.[5] If you have “bugs”[6] in your program, try to find them yourself rather than immediately asking for help. Even though I enjoy helping students, it gets a bit tiresome when someone insists that “the program you gave us doesn’t work” after about 10 seconds of effort and then expects me to fix it. You’ll learn much more if you spend the time to find and repair your own mistakes.

Section 2.4 Your Expectations

You should have some realistic expectations for the class. You will most likely NOT be recruited by Electronic Arts[7] following the completion of this course. You will be able to create some interesting interactive computer generated graphics, however. You will also have a fairly solid background in basic programming and a platform for further learning, either independently or formally. Since this course was started, over 40 former students have gone on to careers in the computer science field. So if this work interests you, then you may have found the start to a great career opportunity!

Section 2.5 Justifications

Back in the early days of our computer science curriculum, graphics programming was synonymous with gaming and we didn't buy those computers so that students could play games.[8] Gradually as time passed and the personal computer world became more graphically oriented thanks to the Mac and Windows operating systems, computer graphics became more mainstream. We began to realize that being able to produce computer graphics meant that not only did you have to understand something about programming, you also had to understand something about mathematics and science if your computer graphics were to look and behave realistically. I am stubborn, though, and it indeed took me two decades to realize that I could teach computer

[4] For Mac enthusiasts, as long as you have Python, PyOpenGL, and a programming editor you should also be able to run the example programs.
[5] In other words, as much as possible do your own work! If you get help from someone or somewhere, and we all do, acknowledge the help. Claiming original work in programs or problem solutions that are actually not your own is dishonest at best
[6] You should look up the history of how programming errors came to be called “bugs”.
[7] A company specializing in interactive sports games.
[8] I remember that one school board member didn't think that we were ready for computers in 1980. I translated that to mean that HE wasn't ready for them. When we purchased the computers, we were expected to compute… whatever that meant, and not "play".

programming utilizing computer graphics. My students seem to enjoy the experience and so do I. I don't know and don't care whether or not this is acceptable practice in other schools… it works for us.

The choice of Python from dozens of other languages is a personal preference. Python is a relatively simple language, but it is easily expanded through external modules such as the PyOpenGL module. Python is object-oriented[9] and is a very popular language. Plus, Python is freely available! The downside is that program execution is somewhat slow[10], but I believe the advantages outweigh the loss of speed. Finally, once you learn Python, it is relatively easy to go on to other languages if you desire (and you may not... Python is a very powerful language!). See section 2.6 below for instructions on how to obtain a copy of Python for your computer.

The choice of which graphics environment to implement was a bit easier. There are two major graphic API's[11] available today, DirectX and OpenGL. DirectX is a windows specific graphics API and OpenGL[12] is available on all platforms. Both are free for programmers to download and use. I personally like OpenGL[13] and think it's a bit better at creating and displaying graphics than DirectX, but I could be wrong.

DrPython is my programming editor of choice. It is freely available and allows interactive access to the Python console. The only minor drawback is that DrPython needs wxWindows to function properly, but that too is a free download. DrPython is written in Python!

There may be other modules or programs that we will use as the course progresses. I will introduce these when the need arises.

Section 2.6 Python Installation

If you are working in the Microsoft Windows environment, I would recommend that you visit http://www.python.org in order to download the latest stable version of Python.[14] You will also need to visit http://pyopengl.sourceforge.net/ to download the PyOpenGL module. In addition, if you wish to install DrPython (recommended), you should go to http://drpython.sourceforge.net/ to download this program editor. There is link on this website to download wxWindows if needed.

[9] Object-oriented programming is a mainstay of modern computer languages.
[10] Python is an interpreted language (for the most part) rather than optimized and compiled like C. This isn't all bad, because it allows us to use Python interactively from a command prompt. Look up interpreted languages for further information.
[11] Computer jargon for a set of commands specific to a particular task, such as graphics.
[12] Originally created by Jim Clark and available on Silicon Graphics (SGI) workstations as GL (graphics language). It used to be a commercial package, but was eventually made available as Open Source software, hence OpenGL.
[13] Prof. Francis was strong influence on my preference. He uses OpenGL/GLUT in his illiMath classes.
[14] I used Python 2.5 and PyOpenGL 3.0.0a under Windows for these program examples. I know that earlier versions of both packages also work properly under both Windows and Linux.

If you are using Linux, you will need to download and install Python, PyOpenGL, and a programming editor on your own. The installation method will depend on the linux distribution (distro) you are running. If you don't have the expertise to do this, there are several tutorials available online or you can ask your favorite computer guru for help.

The examples I use in this text are based on Windows, but I also have a linux system (Ubuntu Linux) on the same computer in order to make certain the programs run properly in both environments.

That's enough for the preliminary information. Let's get ready to program!

Exercises

1. Go to the www.python.org website and spend some time looking at the resources that are available there. You'll find various tutorials and answers to your Python questions. You should bookmark this site in your browser!

2. Go the www.opengl.org website and see what OpenGL is all about.

3. Visit Professor Francis' home page www.new.math.uiuc.edu/~gfrancis and http://new.math.uiuc.edu/ to see how he uses OpenGL in his mathematics classes at U of I.

4. Visit http://pyopengl.sourceforge.net for the PyOpenGL home page. You'll find documentation and the latest Python OpenGL modules here.

5. Visit http://nehe.gamedev.net for an interesting website containing information and tutorials on programming OpenGL using various languages, including Python. The orientation is mostly C/C++, but the Python OpenGL syntax (language grammar) is nearly the same.

6. Visit http://vpython.org for a very nice module for Python. VPython was designed for math and physics instruction and can be used to program the 3D Geowall. Highly recommended!

7. Visit www.pygame.org to see how Python may be used to program games.

8. Finally, use Google or your favorite search engine and look for "OpenGL RedBook". The RedBook is a standard reference for OpenGL and older versions of this fine text can be found online. You can usually download a copy and have it available on your computer desktop for easy referral. I have not provided a link for the RedBook because the links have a tendency to change over time. If you can also find a copy of the OpenGL BlueBook, so much the better. The "Blue Book" is a reference for specific OpenGL commands and it may be possible to find an older version online.

Note: Throughout this text you will be asked to type in programs from the chapter sections as well as create some programs of your own. A few comments are in order. First, all program code listings in this text will be written using the **`Courier New`**

`(bold)` font. Second, some program lines may be placed in quotes such as "**`for t in arange(0.0, 3.14, 0.01)`**" Do NOT type the quotes in the program unless specifically requested to do so. Quotes are usually used only in **`print`** statements. Finally, I suggest that you type the programs in each section of text exactly as written as a starting point. Once the program is running properly and it is saved for future reference, then you should feel free to experiment with the code according to the demands of the exercises at the end of each section or according to your personal tastes and creativity. Make certain that you save any modifications to the original code under a NEW name so that you can preserve the original for use in future exercises.

Enjoy!

Chapter 3 Your First Python Program

Section 3.1 Super-3 Numbers

We are going to start programming by using a numerical example. This is the only program in this text that will not generate graphics, so please don't panic. I'm going to use this program to illustrate some basic programming concepts and hopefully to show you how powerful computers can be.

Super-3 numbers are numbers that when first cubed and then multiplied by 3, result in a value that contains three "3's" in succession.[1] For example, the first Super-3 number is 261, because when we multiply the cube of 261 by 3, we get 5$\underline{333}$8743. How many other Super-3 numbers are there less than 1000? How about less than 10000? Do you want to try to find these numbers by hand? I didn't think so. Let's see if Python can help.

First, start DrPython. You should see a screen similar to Figure 3.1 below.

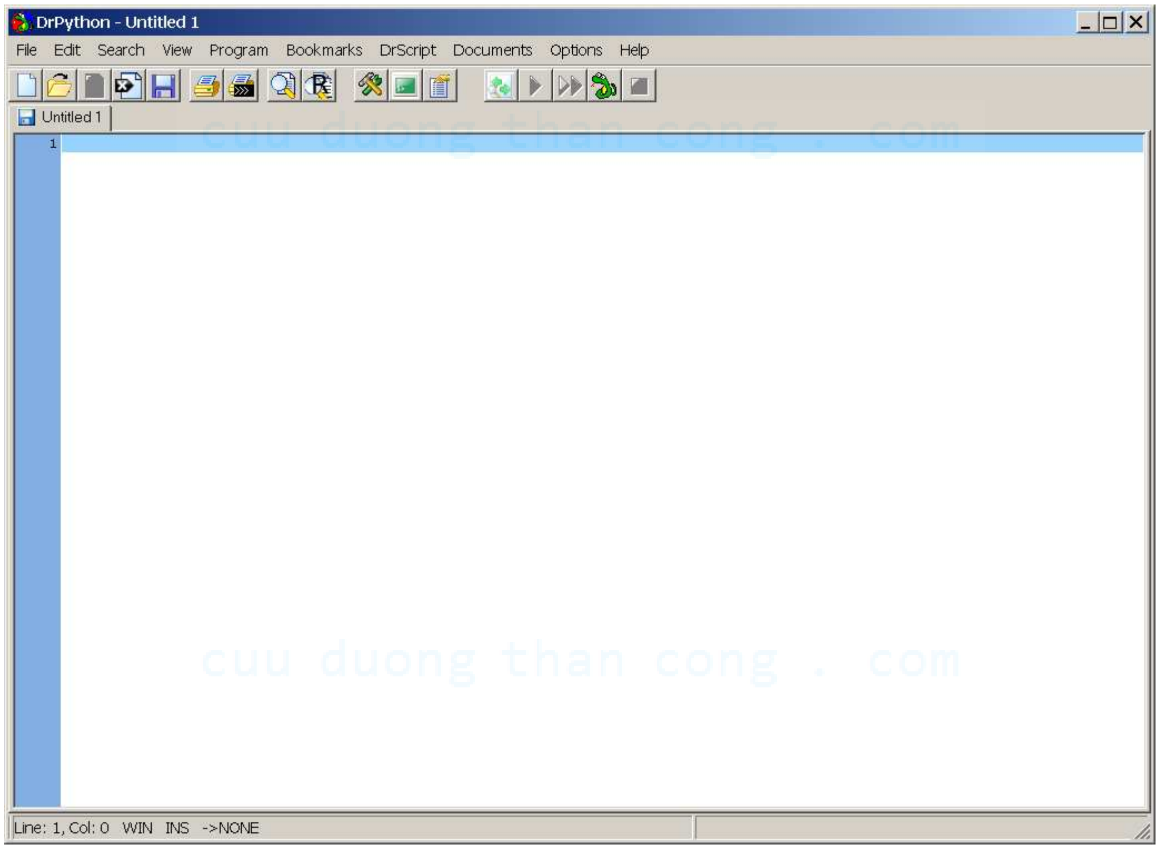

DrPython - Untitled 1
File Edit Search View Program Bookmarks DrScript Documents Options Help
Untitled 1
Line: 1, Col: 0 WIN INS ->NONE

Figure 3.1

[1] See Pickover, C. A. (1995) "Keys to Infinity", New York: Wiley p. 7

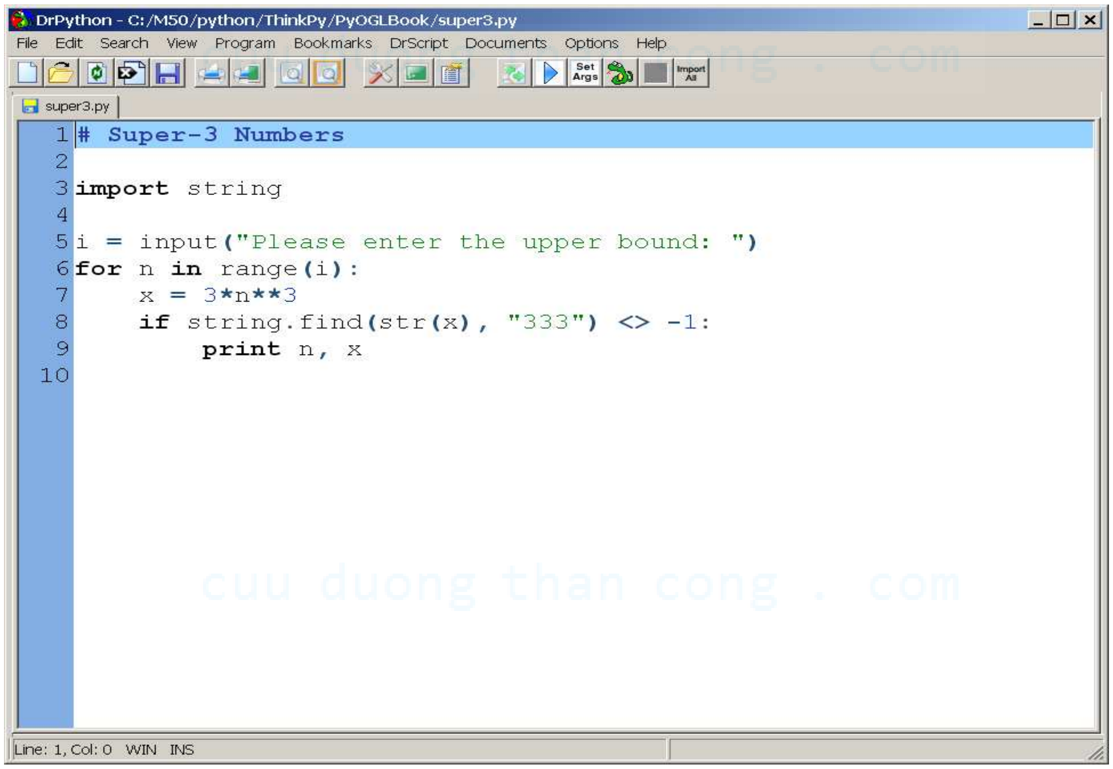
If a blinking cursor isn't visible in the white workspace area in next to the "1", simply click in the workspace area and the cursor should then appear.

Now type the following program exactly as it is listed below.[2] Make certain that you indent each line as shown in the program listing!

```
# Super-3 Numbers

import string

i = input(“Please enter the upper bound: “)
for n in range(i):
      x = 3*n**3
      if string.find(str(x), “333”) <> -1:
            print n, x

# End of program
```

Notice that DrPython automatically numbers lines for us. This feature is very handy when we are trying to trace program errors.

Your screen should look similar to Figure 3.2:


DrPython - C:/M50/python/ThinkPy/PyOGLBook/super3.py
File Edit Search View Program Bookmarks DrScript Documents Options Help
Set Args
Import All
super3.py
Line: 1, Col: 0 WIN INS

```
# Super-3 Numbers

import string

i = input("Please enter the upper bound: ")
for n in range(i):
    x = 3*n**3
    if string.find(str(x), "333") <> -1:
        print n, x
```

Figure 3.2

[2] Program listings will use the **Courier New** font throughout the text.

Once you have typed in this short program and have checked it twice[3] for accuracy, save it to your program folder by clicking on “File” and “Save As”. In the dialog box that appears, type “**super3.py**”[4] for the name of the program (do not use quotes) as shown in Figure 3.3. Click “Ok” when you have finished. Note that your directory and the saved programs that appear will be different from mine.

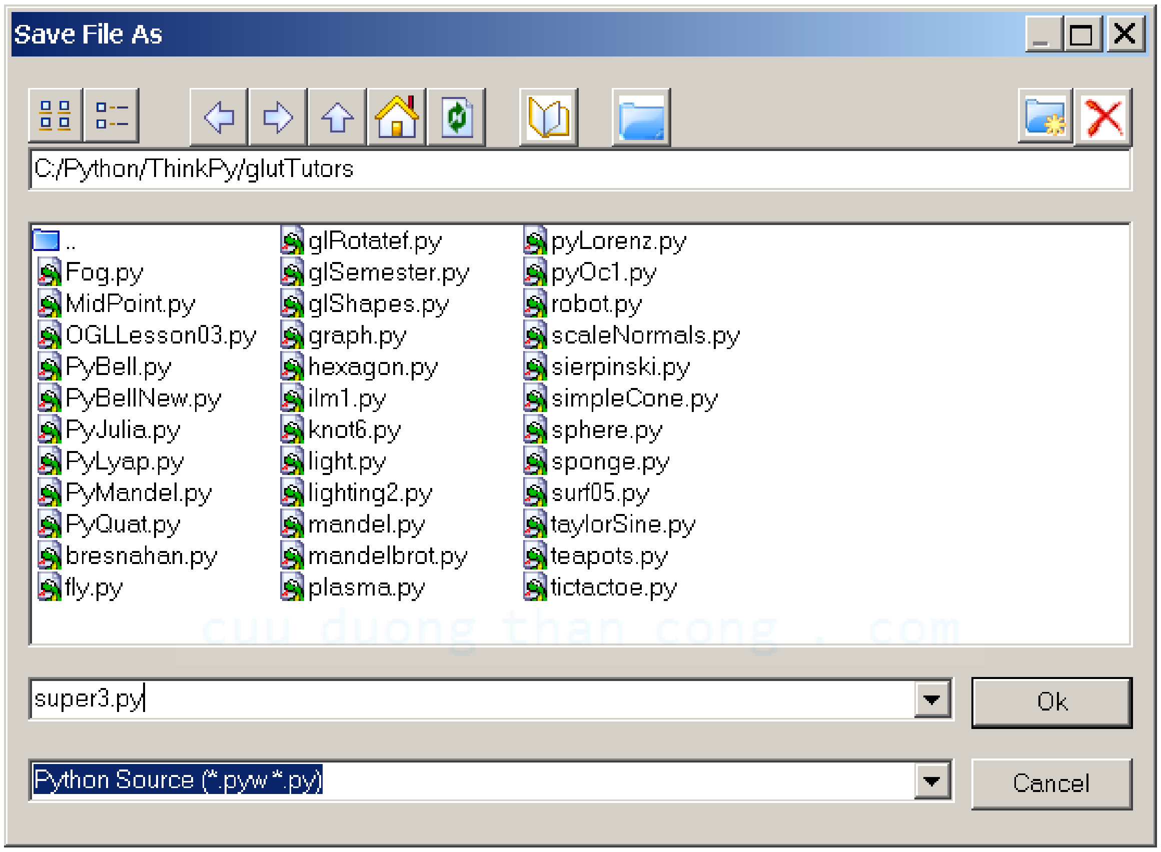
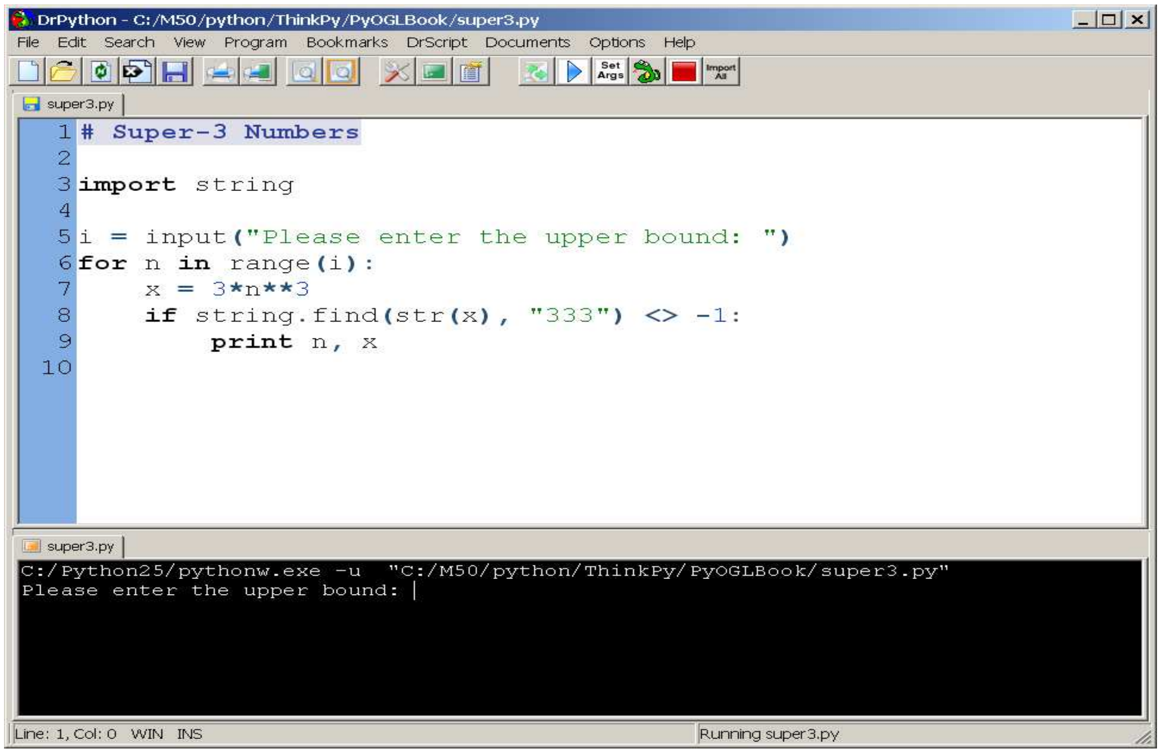


Figure 3.3

When the program has been saved, let’s run it and see what happens! Click on the white “Play” triangle in the menu area. You should see something similar to Figure 3.4 if your program contains no errors. Notice that there is a new area below your program listing. This is the console area and all error messages and results of program calculations will be found here. If everything is OK and you have a blinking cursor in the console, you should be able to type in a value, press enter, and the program will run. Type in 1000 and press enter. You should see the Super-3 numbers less than 1000 displayed in the console as shown in Figure 3.5.

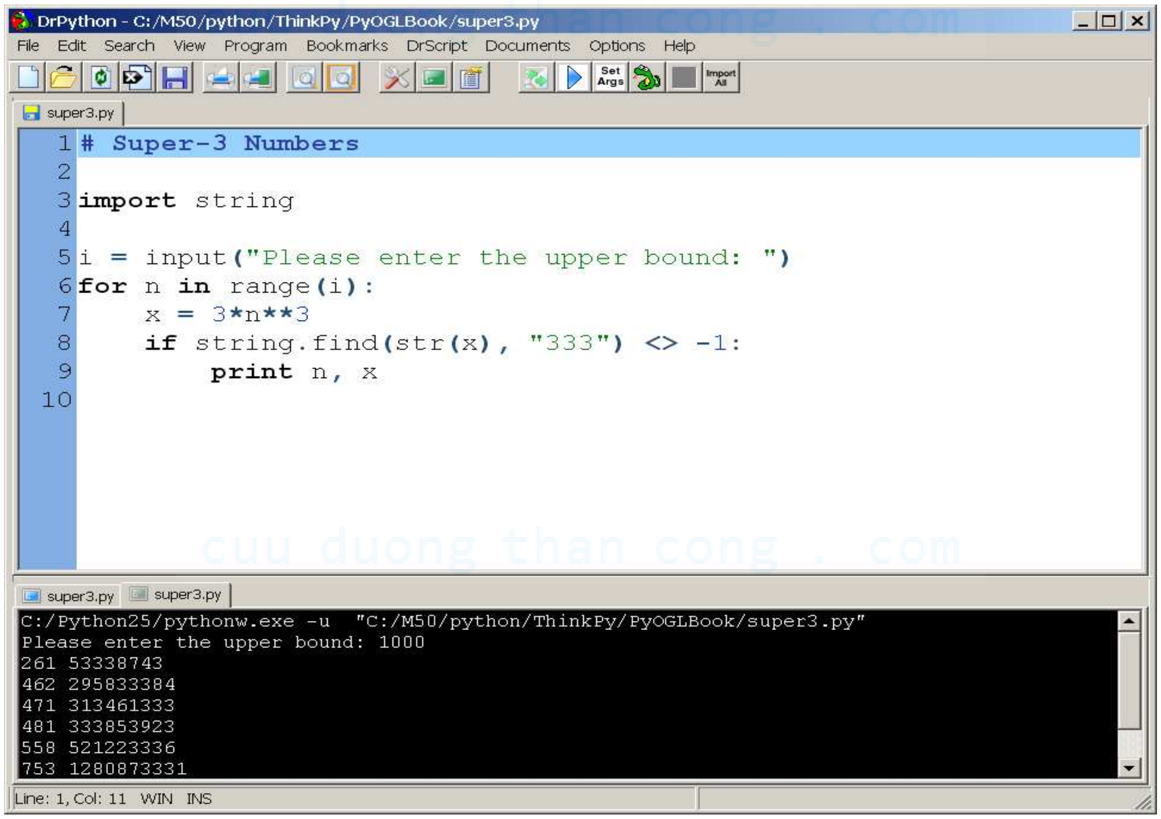
[3] OK, three times. I'm not kidding.
[4] The ".py" ending (suffix) tells the Python interpreter that this program is a Python program and should be treated as such. Failure to add the ".py" suffix will probably cause either unpredictable results or the program will not run at all.


DrPython - C:/M50/python/ThinkPy/PyOGLBook/super3.py
File Edit Search View Program Bookmarks DrScript Documents Options Help
super3.py
Line: 1, Col: 0 WIN INS
Running super3.py

```
# Super-3 Numbers

import string

i = input("Please enter the upper bound: ")
for n in range(i):
    x = 3*n**3
    if string.find(str(x), "333") <> -1:
        print n, x
```

```
C:/Python25/pythonw.exe -u  "C:/M50/python/ThinkPy/PyOGLBook/super3.py"
Please enter the upper bound: 
```

Figure 3.4


DrPython - C:/M50/python/ThinkPy/PyOGLBook/super3.py
File Edit Search View Program Bookmarks DrScript Documents Options Help
super3.py
Line: 1, Col: 11 WIN INS

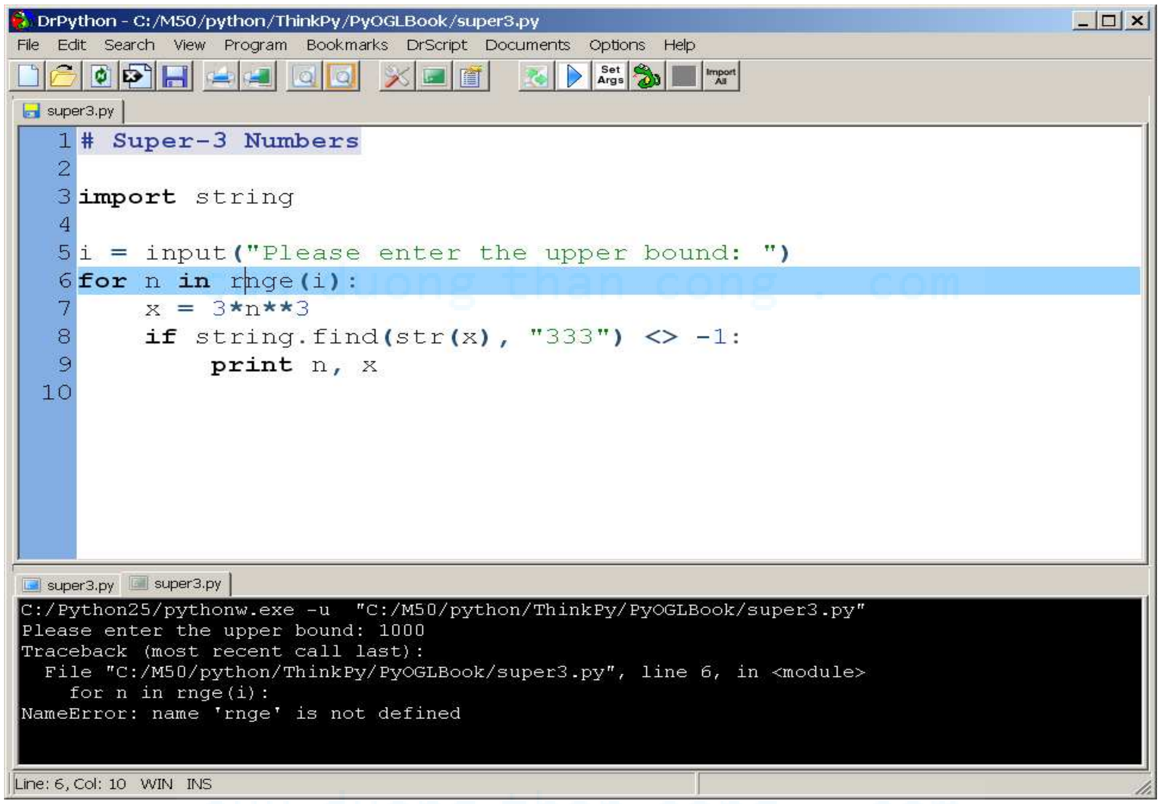
```
# Super-3 Numbers

import string

i = input("Please enter the upper bound: ")
for n in range(i):
    x = 3*n**3
    if string.find(str(x), "333") <> -1:
        print n, x
```

```
C:/Python25/pythonw.exe -u  "C:/M50/python/ThinkPy/PyOGLBook/super3.py"
Please enter the upper bound: 1000
261 53338743
462 295833384
471 313461333
481 333853923
558 521223336
753 1280873331
```

Figure 3.5

Try running the program again. This time enter 10000 or 100000. What do you see? If you want to stop a calculation at any time, press the white square "Stop" button to the right of the green Python icon.[5]

If an error occurs, carefully read the message(s) displayed in the console. The message or messages will usually give you some indication of the problem.[6] Look at Figure 3.6 for an example of an error message. Notice that Python is telling us that in line 6, the name "**`rnge`**" is not defined. Try it! Change the spelling of **`range`** in line 6 to **`rnge`**, save, and then run the program again. What happens? If you change the spelling back to **`range`**, the program should work properly once more. Many program errors are simple spelling errors and are easily remedied. We call such spelling errors "syntax errors". Computer languages such as Python require a very precise usage of commands. Make certain that you type all commands correctly! However, don't feel too badly when errors occur. Even the best programmers make mistakes. Just be prepared to hunt down programming errors ("bugs") and fix them!


DrPython - C:/M50/python/ThinkPy/PyOGLBook/super3.py
File Edit Search View Program Bookmarks DrScript Documents Options Help
Set Args
Import All
super3.py
super3.py
super3.py
Line: 6, Col: 10 WIN INS

```
 1 # Super-3 Numbers
 2
 3 import string
 4
 5 i = input("Please enter the upper bound: ")
 6 for n in rnge(i):
 7     x = 3*n**3
 8     if string.find(str(x), "333") <> -1:
 9         print n, x
10
```

```
C:/Python25/pythonw.exe -u  "C:/M50/python/ThinkPy/PyOGLBook/super3.py"
Please enter the upper bound: 1000
Traceback (most recent call last):
  File "C:/M50/python/ThinkPy/PyOGLBook/super3.py", line 6, in <module>
    for n in rnge(i):
NameError: name 'rnge' is not defined
```

Figure 3.6

[5] The white (or red in newer versions) stop button can be used to halt a program while it is running. If DrPython doesn't behave as it should or a program fails to run, check to see if this button is "available" (white). If so, press it to stop whatever is in progress and hopefully return everything back to normal.

[6] **NOTE****** Sometimes an error message will be generated and the line that is "flagged" appears to be OK. If this is the case, look at the first unremarked line ABOVE the indicated line and see if there is an error with that line. You may have forgotten to close a parenthesis? Also, indentation is extremely important in Python as we will soon see. If you do NOT indent properly, your program will either not run at all or will run in unexpected ways.

Let's go over our program word for word and see if we can understand it. I'll type the program lines in bold and in **`Courier New`** font for emphasis. Here's the first line:

```
# Super-3 Numbers
```

Any line beginning with a "**#**" sign is a remark statement. Remarks are ignored by Python, but serve as valuable comments and reminders, particularly in long programs. Do NOT skimp on remark statements. Use them wherever and whenever you need to make a note to yourself (or others) about what your program is doing[7], the meaning of the variables you are using, and/or your intentions at that point in the program. Now the next line:

```
import string
```

"**`import`**" statements bring in new commands that you can use to extend the Python language. They are almost always placed at the beginning of the program. The **`import string`** command adds some neat string handling commands for our programming pleasure. In Python, a string is any chain of letters or a mixture of letters and numbers.[8]

```
i = input("Please enter the upper bound:  ")
```

When Python encounters an **`input()`** statement, it stops and waits for the user to type something. Here, whatever the user types is stored in the variable **`i`** for later use when the **Enter** key is pressed.[9] The "**`=`**" sign acts as an assignment statement in this line of code. Note that whatever you type between the **`" "`** is printed on the console screen.

```
for n in range(i):
```

Computers are excellent at repetitive tasks. They never get tired! A **`for`** statement is a loop. A loop is a repetitive process a bit like a Ferris wheel… it goes around and around as many times as we specify. In Python, the <u>indented</u> statements below the **`for`** statement will be looped or iterated.[10] In this case, we will loop through those indented statements as many times as the value we entered for **`i`**. If we entered 1000 for **`i`** in the **`input`** statement, the variable **`n`** will take on ALL the values from 0 to 999[11] and we'll instruct the computer to examine every number in this range to see if it's a Super-3 number. How does the computer examine the numbers? Look at the next line of Python code:

```
        x = 3*n**3
```

[7] You skip using remarks at your own peril. If you program anything of any complexity and try to figure out later what you've written, the program may as well have been written by someone else!
[8] "Hello" is a string. 1234 is a number. "R2D2" is a string. 3.1415 is a number.
[9] We can use any variable we wish as long as we don't use a word that Python already knows.
[10] The word "iteration" is synonymous with repetition or looping.
[11] Those crazy computer scientists. They begin counting with 0. Go figure.

This statement takes the values we generate for **n** (0 to 999) and one at a time will cube each of those values and then multiply each cubed value by 3. The **`n**3`** is the cubing process, so you can infer that the double asterisk (**`**`**) raises a number to a power. The single asterisk (**`*`**) is the multiplication sign. Just as in algebra, the power is applied first, followed by the multiplication.[12] The results of the calculation **`3*n**3`** is then stored in the variable **x**. As mentioned in the footnote, there is nothing sacred about the use of the variable **n** in this program. We could just have easily used **testnumber** or **super3candidate**[13] as variables instead of **x** and **n**. Usually you choose variables with meaningful names and if needed, use remark statements to remind everyone how the variables are being used. Anyway… at this point, **x** contains the value of our calculation.

At this point, we need to see if **n**, whatever it is, is a Super-3 number. How do we do this? We know that a Super-3 number is an integer that when cubed and multiplied by 3 results in a value with a "**333**" somewhere in the number. We could look at every number visually and decide for ourselves whether or not it fits the Super-3 conditions, but that would be tedious (and silly). We'll let the computer make the decision for us. Computers are excellent at making decisions, but we must first formulate the decision statement properly. Look at the next line:

```
if string.find(str(x), "333") <> -1:
```

Decision or conditional statements usually start with **`if`** and every indented line under the **`if`** statement is included in the conditional statement or block. Look at the following pseudo-code example:

```
if statement_true:
      do something remarkable
```

Generally the meaning is as follows: "if the statement is true, do something. if it's false, ignore me". An **`if`** block of code makes decisions! You may have been warned as a child to "Look both ways before you cross the street!". Well, that is certainly an important instruction, but that alone is not enough to insure our safety. Somewhere in our brain we must have an **`if`** block of code such as:

```
if car_is_approaching:
      stay on the sidewalk
```

Think about this! How important are such decisions in our own life? Really, computers are nothing more than machines with the capability of looping endlessly through code, making as many decisions as we care to write.

Now back to our **`super3.py`** program. Let's take this **`if string`** statement apart piece by piece. The **`string.find`** command is one of the neat string commands we

[12] The order of arithmetic is the same in algebraic programming as in your math class. Please Excuse My Dear Aunt Sally… parentheses, exponentiation, multiplication, division, addition, subtraction.
[13] Python distinguishes between upper and lower case variables. The variable **'Cat'** is different from the variable **'cat'**. Be careful with your spelling!

can use after the **`import string`** statement. It allows us to search strings for patterns of letters, symbols, or numbers. In parentheses, we see **`str(x)`**, which temporarily converts the number stored in **`x`** (calculated from the previous line of code) to a string so we can search it. **`“333”`** is the pattern we are trying to match. If a **`“333”`** pattern is NOT found in **`x`**, the **`string.find`** command returns (or equals) **–1**. So, in English, this statement says: **if** in **`str(x)`** we find a **`“333”`** pattern then **`string.find`** will **NOT** equal **–1** (“**<>**” means NOT equal... we could also use "**`!=`**". Try it!) and that would make the statement TRUE (**`string.find`** is indeed NOT equal to -1). If in **`str(x)`** we DON’T find a **`“333”`**, that means that **`string.find`** equals –1 and the statement is FALSE.[14] Remember that a FALSE statement means that we will ignore any indented code in **`if`** block. The colon symbol “**`:`**” finishes the “**`if`**” statement.

You may be wondering what happens if the **`string.find`** function actually does find the search string **`"333"`**? Does the function return a value other than **`-1`**? Well, obviously it does or our program would not work! But what value does the **`string.find`** function return if it finds a match and why can't we use THAT value in our **`if`** statement (**`if string.find(str(x), "333") == value:`**)? It turns out that the **`string.find`** function does not return the same value every time the search string is matched. The value the function returns is the location or index in the string (or number converted to a string in this case) of the first character in the search string. For example, the first Super-3 number is 261. This translates to a Super-3 value of 53338743. The first "3" of the **`"333"`** search string in this number is located in the "1" location... remember that we start counting from 0 in computer science; the first number is in position 0! So, **`string.find`** would return a value of 1 for the number 53338743. The second Super-3 number is 462, which has a Super-3 value of 295833384. For this value, **`string.find`** would return a 4, indicating the position or index of the first character of the search string, again starting from 0. So we can't simply check for a single value for **`string.find`** in the **`if`** statement to indicate that a match has been found. We'll explore this concept further in the exercises at the end of this section.

If the statement in the **`if`** line is TRUE, then any indented[15] lines immediately below it are executed (not shot… but implemented by the Python interpreter). The single indented line is:

```
print n, x
```

This line prints both the Super-3 number **`n`** and it’s value **`x`** after cubing and multiplying by 3 (so you can see for yourself the **`“333”`** in the number). Remember, this line is only “executed” in the event that that **`if`** statement above it is TRUE.

Well, that’s all for the first program. In the next chapter we’ll being working with graphics. First, though, we are going to try some exercises. These problems will require you to modify the program we’ve just discussed. If you want to save your original code,

[14] Think about this! It is easy to become confused with conditional **`if`** statements.

[15] Indented lines are crucial in Python. Most other languages ignore indents and they are used to make code easier to read. In Python, indents are part of the language and MUST be used appropriately. Note the indented lines after the **`for`** statement. All those lines are in the **`for`** loop. Likewise, the indented lines immediately after the **`if`** statement are part of the **`if`**

you'll need to name your newly modified program something different. Suggestions for names might be **ch3ex1.py** for Chapter 3 Exercise 1… but you are free to name the modified programs anything you or your instructor wish.[16]

Section 3.2 Conclusion

Computers are great at handling input, doing repetitive tasks, producing calculations, making decisions, and displaying output. The simple Super-3 program we used as an example in this chapter demonstrated the power of a computer in performing all of these operations in just a few lines of code. How long do you think it would take you to find all the Super-3 numbers less than 10000 by hand?

Exercises

1) Super-d numbers are a more general case of Super-3 numbers.[17] With Super-d numbers, you replace d with the number of your choice. For example, Super-4 numbers would be those numbers when raised to the 4th power and multiplied by 4, contain "4444" somewhere in the calculated value. How would you modify the Super-3 program to find Super-4 numbers? What is the smallest Super-4 number?

2) Are there any Super-5 numbers less than 10000? What is the smallest Super-5 number? Remember to search for "55555" in the "`if`" statement!

3) Can you modify the program to search for other patterns? How about the pattern "1234"?

4) What happens if you change the formula in line 7 to something other than the Super-d format? You won't be searching for Super-d numbers, but perhaps you will find something interesting? Feel free to explore a bit! Search for strange patterns! For example, search for "314159".

5) This is a thought question. How many Super-3 numbers are there? Could you use a computer to find them all?

6) Rewrite the Super-3 program to check for a match using the "`>`" operator or the "`>=`" operator.

7) Add some statements to the program to print the values returned by `string.find` when the function actually finds a match.

The following exercise is a bit more difficult.

8) Python has other methods of looping such as the statement `while`. Research other looping methods for Python and see if you can rewrite the program using a different looping structure. Remember that indentation is important! Also, you will probably

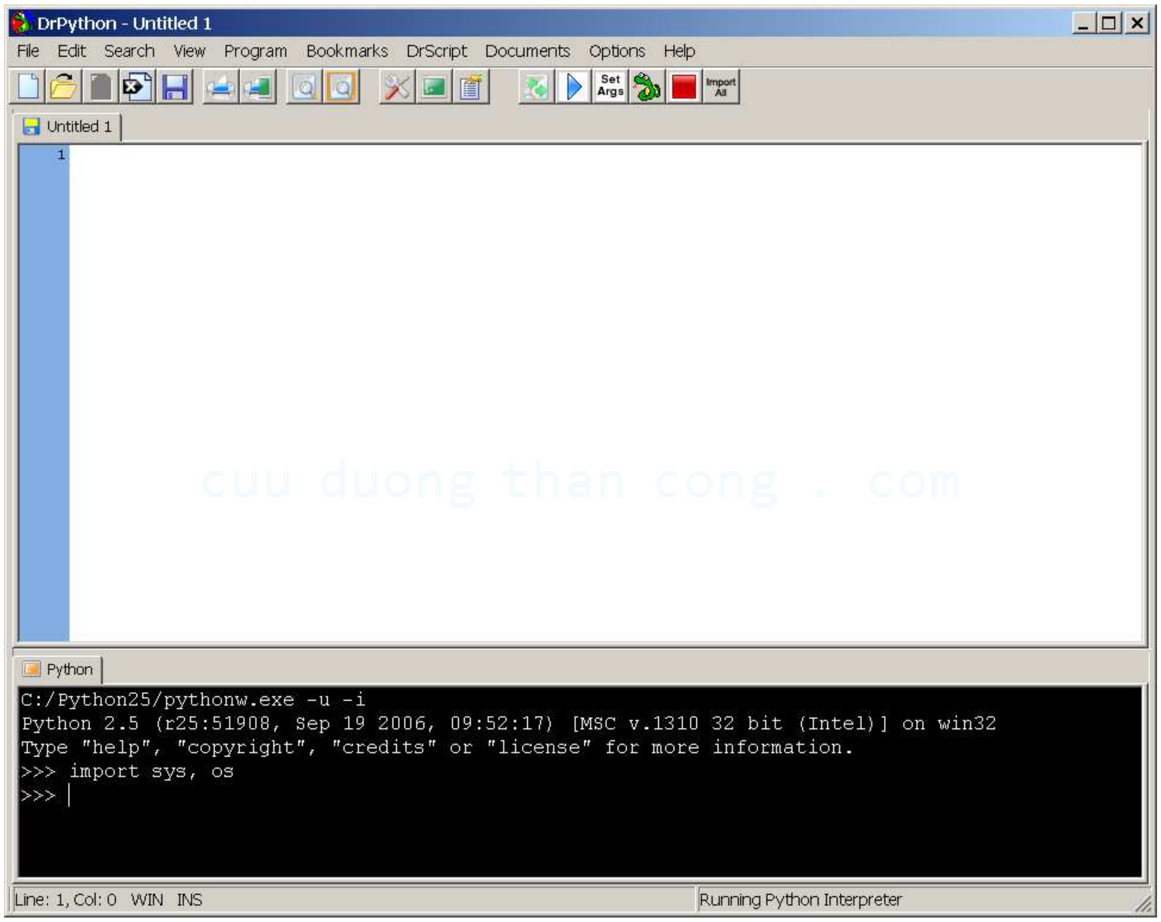
[16] But please don't forget the ".py" suffix! I know, I already told you this but it's really important.
[17] http://mathworld.wolfram.com/Super-dNumber.html

discover that you need to be able to find a way to count or increment the value of the variable you are using to construct Super-3 numbers. How can you do this? Do you need to tell Python that the variable is going to store a number rather than a string? How would you do this?

Chapter 4 Your First OpenGL Program

Section 4.1 The Interactive Python Interpreter

Before we begin with OpenGL, I think it's appropriate to show you a feature of Python that we haven't yet discussed. Start DrPython (if it isn't already open) and press the "green Python" button to the right of the program start icon. Your screen should look something like Figure 4.1.


DrPython - Untitled 1
File Edit Search View Program Bookmarks DrScript Documents Options Help
Set Args
Import All
Untitled 1
Python
Line: 1, Col: 0 WIN INS
Running Python Interpreter

```
C:/Python25/pythonw.exe -u -i
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys, os
>>> 
```

Figure 4.1

Note the ">>>" in the console area. This prompt tells you that you are in the interactive Python mode. You can type Python commands at this prompt and they will be immediately executed. For example, type **`3+7`** and press enter. It seems Python knows a little arithmetic! Now type **`10/5`** and press enter. Of course, Python tells you the answer is 2. Now type **`10/4`** and press enter. What gives? Python still answers with 2! The reason for this behavior is that Python will perform arithmetic according to the numbers we give it. If we supply only integers in our problem, the results of Python arithmetic will be an integer. Try **`10/4.0`** or simply **`10/4.`** and see if you get the answer you expect. As long as one of the numbers we use in the calculation is a floating point or decimal value, then Python will respond with floating point calculations.[1]

Python arithmetic may seem quirky, but once you have some programming experience, it isn't difficult. Just remember the integer vs. floating point format to avoid program calculation errors. You might try multiplying, adding, subtracting, and raising some numbers to a power to see if you get the results you expect.

Before we close the interactive session, type **`print 'hello'`** and press enter. No surprise, right? Now type **`name = 'Dubbya'`** and press enter. Nothing happened! Well, actually something DID happen… what was it? Type **`print name`**, press enter and see if you were correct. Yes, the string '**`Dubbya`**' was stored in the variable **`name`** and we can view the variable contents by using **`print`**.

The interactive mode can be useful to perform quick calculations or to check simple program structures. Before we exit, notice in the message in the lower right corner of the DrPython window. It should say "Running Python Interpreter". This tells us that we are in the interactive mode. Exit the interactive mode by clicking the red (square) stop button and then clicking on the green monitor button to the left of the program start icon.[2]

Section 4.2 Introducing Python OpenGL

We are ready for our first Python OpenGL program. Start DrPython[3] and type in the following lines in the upper programming area, starting with line 1:

```
#  First Python OpenGL Program
#  ogl1.py
```

What do these lines do? Remember, they are remark or comment statements, so they really don't DO anything other than provide notes or reminders for you and anyone else who might read your code. I can't emphasize enough the importance of using comment statements in your code, particularly as the complexity and length of your programs increase.

Now, save the work in your directory as you have done previously (using "File" and "Save As") and giving it a name such as **ogl1.py**. It's a great idea to save your work every few lines or so.[4] Once you have named and saved the program the first time, you can simply push the "diskette" button next to the printer icon to save any new code. You

[1] You should look up floating point, integer, double precision, and single precision numbers. Better yet, visit www.Python.org and search in the online documentation for Python data types.
[2] The green monitor button is useful for closing the console and displaying full screen editing mode. You must click the white stop button to stop the Python interpreter.
[3] You do NOT have to use DrPython as a programming editor. There are others, such as SciTE that are free and also work well. Python comes with its own editor called Idle. You can use that. You can even write Python code in a text processor and run the code from the command prompt by typing: **`C:>\python prog.py`** where prog.py is the name of your saved program.
[4] I once worked for nearly 6 hours on a program and the power went off. I lost everything. I learned to save my work early and often after that.

can see whether or not you need to save your work by looking at the caption bar at the top of the DrPython window. If the words “untitled” or “[Modified]” appear, then save your work!

Let’s add more lines to our program:

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys
```

Remember that Python comes with a basic set of commands that you can immediately use. The **`import`** statements add extra functions to this basic set of commands. With the exception of **`import sys`**, these are **`import`** statements with a somewhat different format than the one used in the first program (**`import string`**). In the first program, we had to preface the “**`find`**” command with the word “**`string`**” in order for it to use the **`string`** module commands (**`string.find`**). In this new **`from... import *`** format, we can use OpenGL module commands directly without having to use the **`OpenGL.GL`**, **`OpenGL.GLU`**, or **`OpenGL.GLUT`** preface in a similar fashion as **`string`** in the Super-3 program. If we simply used **`import OpenGL.GL`**, then every time we used an OpenGL.GL command, we would need to write (for example): **`OpenGL.GL.glVertex2f(x, y)`**. With the **`from... import *`** format, we can simply write: **`glVertex2f(x, y)`**. Much simpler, don't you think?[5] The final **`import sys`** statement provides some housekeeping tools we will need to create the graphics display. We will usually only need a couple of commands from this module, so we do not need to use the **`from... import *`** method here. Likewise, in the Super-3 program we only needed one string function (**`string.find`**). In contrast, we would like ALL the OpenGL GL, GLU, and GLUT commands available to us when we program. Having all the OpenGL commands available will save time!

To summarize, each of the **`from OpenGL`** statements adds new commands to Python. There are OpenGL commands (**`.GL`**), GLU commands (**`.GLU`**), and GLUT commands (**`.GLUT`**). You will know when you are using commands from each specific OpenGL module by looking at the first few letters of the OpenGL command. For example, **`glClear()`** is a command from the main OpenGL (GL) module. **`gluPerspective()`** would be from GLU and **`glutWireTeapot()`** is from GLUT. You will see and use MANY examples of these commands (usually prefaced by **`gl`**, **`glu`**, or **`glut`**) throughout this text. Let's continue:

```
def draw():
	glutWireTeapot(0.5)
	glFlush()
```

This is new. The **`def`** keyword marks the beginning of a function block of code in Python. Functions are segments of code that act as a group or family of commands.[6]

[5] You may have guessed by now that the **`*`** in the **`from... import`** statement imports ALL commands available to that particular module. Good guess!
[6] See the "Odds, Ends, and Terminology" section in this chapter.

The name of this function is "**draw"** and we can call this function from within our Python program by using this name. Note the two indented lines after **def draw():**. The indents mark these lines as belonging to this particular function.[7] The **def draw()** function does two things. First, it uses a **glut** command to draw a wire teapot with a size of 0.5. Second, the **glFlush()** statement "flushes"[8] the drawing to the screen so we can see it. Save your work.

Finally add these lines, making certain to line them up even with the left-hand margin.[9]

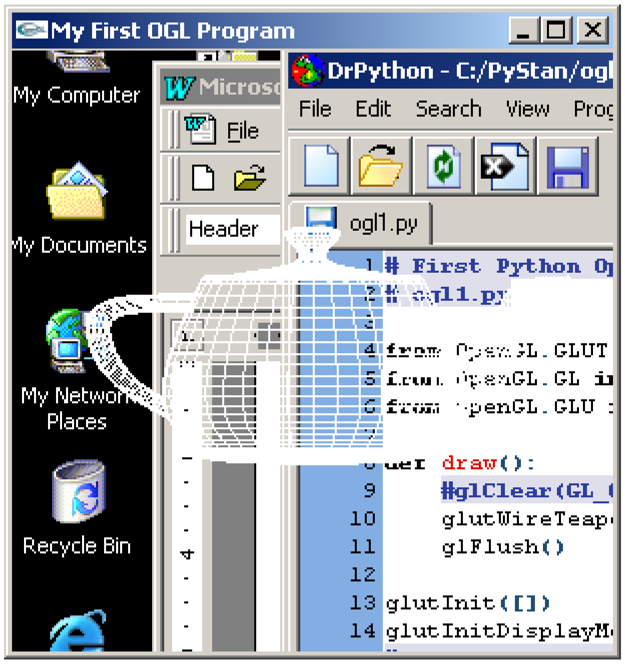
```
glutInit(sys.argv)
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
glutCreateWindow("My First OGL Program")
glutDisplayFunc(draw)
glutMainLoop()
```

Save your work! The first line, **glutInit(sys.argv)** tells Python we are going to be displaying glut style graphics. GLUT stands for "GL Utilities Toolkit" and provides cross-platform (linux-windows-mac) background calculations making the job of doing graphics MUCH easier than if we had to write this code ourselves. In other words, a LOT is going on "behind the scenes" here and GLUT takes care of the gory details. The (**sys.argv**)[10] argument allows us to enter a command line instruction if we wish (we don't in this program). The next line, **glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)**, tells the graphics "wizard" that we are going to be using a single display screen (buffer) **and** that we'll be using the RGB (red, green, blue) color scheme. The vertical line between **GLUT_SINGLE** and **GLUT_RGB** means "and" and is typed using the "shift+\" key (above "Enter" on most keyboards).

The **glutCreateWindow("My First OGL Program")** command uses GLUT to create a graphics window and places "**My First OGL Program**" in the new window's caption bar. The **glutDisplayFunc(draw)**statement tells GLUT where to find the function that creates the graphics scene. In this program, the display function is **def draw()**.[11] Finally, the **glutMainLoop()** statement starts the program running. This is another example of "gluttery" and keeps our program in an eternal loop, allowing GLUT to check continually for things like mouse and keyboard input/control.

[7] A common error in Python is to forget to add a "**:**" after the **def** statement. Ditto for **if** statements and **for** loops.
[8] Yes, I know... like a toilet. Later we will learn how to avoid **glFlush()** by using something called buffering.
[9] Improper indentation can cause major headaches in Python. If you indent these commands, Python will think they belong in the **def draw():** function and the program will not work properly.
[10] Remember **import sys**?
[11] We could have named it anything as long as we are consistent. We could have created a **def picasso():** function that draws the teapot and as long as we used **glutDisplayFunc(picasso)**, everything would be OK.

If you haven't already, run the program! You should see something that looks similar to Figure 4.2 depending on what is in the background on your desktop:



```
# First Python O
# ogl1.py

from OpenGL.GLUT
from OpenGL.GL i
from OpenGL.GLU :

def draw():
    #glClear(GL_
    glutWireTeap
    glFlush()

glutInit([])
glutInitDisplayM
```

Figure 4.2

Well, at least you can see something that looks like a teapot. What happened? When we created the graphics window, we did not specify anything about the background, so GLUT simply copied everything from the screen location into the window and then drew the teapot on top of the whole mess. Move the window around with your mouse (left-click on the caption bar, hold the mouse button down, and move the window). You'll see that the background moves with the window!

We can easily fix this by adding one line of code. Place the statement **`glClear(GL_COLOR_BUFFER_BIT)`**[12] after the **`def draw():`** line. Remember to indent at the same level as the other two lines in this function! Save the program and run it again. You should now see something like Figure 4.3 on the next page. Move this window around with the mouse and everything works as expected. Finally, we are going to add two more lines to our first program. Click on the line that contains the **`glutInitDisplayMode`** command. Press the "End" key on the keyboard or click at the end of this line. Press "Enter" twice to make room for the following two lines of code.

```
glutInitWindowSize(250,250)
glutInitWindowPosition(100,100)
```

Save your work. These lines allow us to specify the size of the graphics window (250x250 pixels on a side) and the initial location of the window (100, 100), which is 100

[12] This statement obviously clears the screen. The default color is black.

pixels to the right and 100 pixels down from the screen origin.[13] If you run the program, you should see a graphics window in slightly different location.

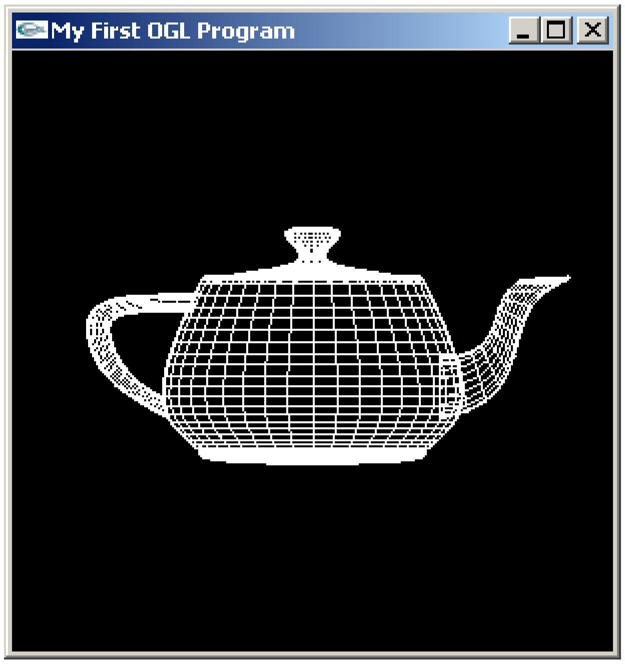


Figure 4.3

Here is the complete listing of our modified first program.

```
# First Python OpenGL program
# ogl1.py

from OpenGL.GLUT import *
from OpenGL.GL import *
from OpenGL.GLU import *

def draw():
       glClear(GL_COLOR_BUFFER_BIT)
       glutWireTeapot(0.5)
       glFlush()

glutInit(sys.argv)
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
glutInitWindowSize(250, 250)
glutInitWindowPosition(100, 100)
glutCreateWindow("My Second OGL Program")
glutDisplayFunc(draw)
glutMainLoop()

# End of program
```

[13] In computer graphics, the origin is by default in the upper left corner of the display screen and uses positive numbers down the y-axis and across the x-axis. You can change this to Cartesian coordinates easily in OpenGL with the origin in the center of the screen. We'll do that a bit later.

Section 4.3 Odds, Ends, and Terminology

Programming can be very frustrating. Computers (at least the ones we use now) are not intelligent. They will do exactly what we tell them to do, but not necessarily what we want them to do… and that's assuming that we have not made any errors in programming syntax or grammar.

There are two major sources of errors in programming. The first concerns visible errors in the spelling of key words, punctuation, and/or grammar. These errors are placed in the category of "syntax" errors. The program simply won't run until all syntax errors are corrected. Fortunately, we are often given information by our programming editor or the Python interpreter about the nature of the syntax error and we can usually fix these errors rather easily. If the syntax error is not found in the line of code specified by the interpreter, it will usually be found in the line immediately preceding the "flagged" line. Often several lines will be listed as having errors. Such a complex listing is called a "traceback" and will include a detailed history of how, where, and in which modules the error occurred. In such instances, look at the very last line in the list of errors. This line will usually contain the offending code and fixing this line may repair the damage.

The second source of errors is more insidious and far more frustrating. Your program seems to run and will not generate error messages, but you don't get the results you expect. This may mean that the program IS running correctly, but you need to rethink your expectations. You should remember that computers will not always do what you want them to do, but they WILL always do what you tell them to do. Such errors are found in the logic of your program and can be very difficult to trace. If you are typing a program from a code listing that is known to work,[14] then you have forgotten to type something correctly. When you find and correct the error, the program will run properly and it may be something as simple as forgetting a line of code or forgetting to use a closing brace in a function. If you are creating your own program from "scratch" or from the **`PySkel.py`** template we'll be using later, then you may have to look more closely. Common problems in Python include improper usage of **`if`** conditional statements, incorrect structure in loops, incorrect indentations in loops, functions, and **`if`** statements, and improper usage of OpenGL commands. Fixing these errors may involve rethinking the problem. Often, using a **`print`** statement to display the values of variables at appropriate times will help immensely. The best weapons against programming errors of any kind are patience, clear thinking, and experience![15]

Terminology is important in any field, not just in computer science. Sometimes fields (such as education!) are unnecessarily burdened with terminology, but for the most part, a properly defined vocabulary unique to a subject area is the most efficient way to teach, learn, and communicate with others in that discipline. Much of the terminology in computer science has become mainstream since the proliferation of computers into nearly every home. There are some vocabulary terms that we need to define, though, to avoid confusion in this course. Teachers tend to repeat themselves (more often as we age…), so I may define some of these terms again (and again) throughout the text. If so, simply nod your head and remember the word(s) and meaning(s)!

[14] Hopefully ALL programs in this text fall under this category!
[15] These are outstanding attributes to have in ANY walk of life.

First, we need to discuss the distinction between functions in computer science and functions in mathematics. In computer science, sections of code that are "set aside" to perform a specific task are called functions (or archaically, subroutines). A function is a code segment that is designed to calculate a value and "return" that value to the main program. Buttons such as "sin()" or "tan()" on a calculator are an example of a function. In Python, we might have something like this:

```
def sqr(x):
     n = x*x
     return n

print sqr(5)

# End Program Listing
```

This small piece of code will actually run![16] We define a function "**sqr**" using **def sqr(x):**. This function takes an argument (a number we send it through code) and stores it in the variable **x**, which is then used in the function by the line **n = x*x**. The **return n** command does exactly that: it returns the value of **n** to the program. In this example, we have defined a squaring function and all we have to do to use this new function is to issue a **sqr(j)** command, where **j** is a number we want to square. So, a function in Python returns a value. A function can also be a code segment that is designed to perform a task, but it does not necessarily perform a calculation (again, archaically called a subroutine). The **def draw():** function in the program in this chapter is such an example. In modern terminology, all blocks of code that are set aside to perform a task are called functions.

Functions in mathematics are analogous to functions in programming. A function in mathematics is an operation, action, or process that converts one or more numbers into another (probably different) number.[17] The key to a mathematical function is that when we supply a number or numbers as input, we will get only one unique output.[18] You might think that this definition is restrictive and that we will be unable to produce plots of 2D curves such as circles (which fail the vertical line test) and 3D objects such as spheres (ditto). The short answer is that we won't use a single function for these objects. We will use combinations of equations in parametric or polar form to produce any plot we choose.

Another important term is the concept of *iteration*. Iteration is the process of repeating the same calculation(s) a specified or perhaps indeterminate number of times using a loop structure. Usually the repeated calculations are performed on a mathematical function or set of functions. One key to iteration is that the result of the previous computation is used as the input for the next computation. This is analogous to starting with a value and repeatedly pressing the "sqrt()" or "sin()" button on a calculator.

[16] Use "File|Save As" and name the program something like "sqr.py".
[17] Devaney, Robert L. (1990). "Chaos, Fractals, and Dynamics: Computer Experiments in Mathematics." Addison-Wesley Publishing Co.
[18] Hence the vertical line test used on graphs in algebra.

You may encounter these and other terms again as we journey through this text. If it sounds as if I'm defining for the first time a previously defined term or phrase, simply bear with me. The more times you read a term and perform a task, the more likely you are to remember the term and learn the task. Also, I'm a teacher (and I'm old) and I tend to repeat things repeat things.

Section 4.4 Conclusion

In this chapter we introduced both the Python interactive interpreter and our first OpenGL based program. We also briefly discussed programming errors. Always remember that you MUST be careful when programming. As I stated earlier, a Python program, or any program in any language, will not always do what you want it to do, but it WILL always do what you TELL it to do. Pay strict attention to both syntax errors and logical errors! Be prepared to exercise patience and thought when attempting to fix programs that are not running correctly or perhaps not running at all!

We also introduced some terminology that we'll be using throughout the text. This terminology should not be considered final or complete. We will probably add to your vocabulary as we progress in knowledge and no doubt many of the terms will be visited more than once!

The following exercises are designed to allow you to explore the code in our first OpenGL program. You probably want to keep your original program intact, so open it again (if it isn't open) and use "File" and "Save As" to save it under a new name (ending in ".py"). You can then experiment with the new program. Also, you'll notice that most programming editors such as DrPython can have several programs loaded at once. Under the menu labels or icons, you should see a tab or tabs which contain the names of the program or programs currently open. You can click on any of the tabs to make that particular program active and ready to edit/run.

Exercises

1) Change the caption to display your name instead of "**`My First OGL Program`**".

2) Change the initial window location to something other than (100, 100). Can you predict where the window will appear based on the ordered pair of numbers?

3) Change the initial window size to (400, 400). Try various values and see what happens. The numbers do not have to be equal, although you should probably try to avoid negative numbers and/or zero. Try a tall narrow window or a short wide display. Which number controls the width and which the height?

4) Change the size of the teapot from (0.5) to larger and smaller values. What happens?

5) Comment the **`glutWireTeapot`** command to disable it (How?). Create a blank line below it and add this line: **`glutWireSphere(0.5, 10, 10)`**

a) Change the 0.5 to 0.75 and see what happens. What does this value do?
b) Change the line to: **`glutWireSphere(0.75, 25, 25)`**
c) What do the last two values do?
d) Change the line to: **`glutWireCube(1.0)`**
e) Change the line to: **`glutWireTetrahedron()`**
f) Change the word **`Wire`** to **`Solid`** in each of the commands above.

There are additional GLUT geometric shapes that we'll explore later.

6) Research "iteration". How is it used in computer science? Are there particular problems that can be more easily solved by iteration than by other means? Also research "recursion". Are iteration and recursion the same? If not, how are they different?

7) Research "function" and "subroutine". Compare the definitions you find with the definitions given in this text.

8) You'll need a calculator for the next few exercises. Type in a number larger than 1 (such as 5). Repeatedly press the "sqrt()" button. Do you eventually reach a value that doesn't change? What is this value? Do you think this is correct? Try another value > 1 and see if you get the same result.

9) Now type in a positive[19] value less than 1. Repeatedly press the "sqrt()" button. Do you eventually reach a value that doesn't change? What is this value? Again, do you think this is correct? Try another positive value less than 1 and see if you get the same result.

10) Type in another value and repeatedly press the "cos()" button. Do you eventually reach a value that doesn't change? What is this value? This value is the solution to what function? Try another number and see if you get the same result.

11) Try the "sin()" and "tan()" buttons in the same manner. Do you get the same results? Why or why not?

12) Try running the program from this chapter again, but this time, use (.5) instead of (0.5) for the size of the teapot. What happens? Do we need to include the leading zero to the left of the decimal? Some languages require a leading zero in front of decimal values between -1.0 and 1.0.

[19] Why a positive value?

Chapter 5 2 Dimensional Graphics

When you stop and think about it, all computer graphics are 2 dimensional.[1] We can create the illusion of 3D (and beyond) by the proper placement of pixels, objects, and the use of visual cues to "trick" your mind into seeing depth, but essentially we are still plotting points on a 2D screen. With this concept in mind, it makes sense to start with "true" 2D graphics and we'll begin by plotting points.

Section 5.1 Plotting Points

One of the basic functions of any graphics language is to manipulate individual points or pixels within the graphics window. We need to have complete control over where the points are plotted, the point size, and the point color. If we can control the plotting and characteristics[2] of individual points, then there is nothing we can't draw!

Our first program in this chapter will be a simple exercise in setting up the graphics window so that it behaves much like the coordinate system you learned in algebra class. As stated in an earlier chapter, the origin in a graphics window is usually in the upper left hand corner. We would like to move the origin to the center of the graphics display window. Once we have established the origin, it should be simple to plot points where we please.

Start DrPython or your Python editor and enter the following program. Make certain that you save the program in your directory with a "**.py**" suffix or ending. I'll suggest a name for each program in the first remark[3] statement, but feel free to choose your own name if you wish. Remember to pay strict attention to indenting!

```
#  PyPoints.py
#  Setting a coordinate system with central origin

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys

def init():
     glClearColor(0.0, 0.0, 0.0, 1.0)
     gluOrtho2D(-1.0, 1.0, -1.0, 1.0)

def plotpoints():
     glClear(GL_COLOR_BUFFER_BIT)
     glColor3f(1.0, 0.0, 0.0)

     glBegin(GL_POINTS)
     glVertex2f(0.0, 0.0)
```

[1] Unless you have a holographic display? Do you? Guess what I want for Christmas?
[2] Sometimes called "attributes"
[3] Since I am explaining the first programs in great detail, I'll keep remark statements out of the program listing at this point in the text in order to avoid clutter.

```
     glEnd()

     glFlush()

def main():
     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB)
     glutInitWindowSize(500,500)
     glutInitWindowPosition(50,50)
     glutCreateWindow("Plot Points")
     glutDisplayFunc(plotpoints)

     init()
     glutMainLoop()

main()

# End of Program
```

When you run the program, assuming you have no errors, you should see a graphics window with a single tiny red dot in the center (at the origin!). Not very exciting (yet), but it's an important "point".[4]

Let's look at the listing in detail. The first few lines we've seen:

```
#  PyPoints.py
#  Setting a coordinate system with central origin

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys
```

As explained previously, the lines following the remark statements add OpenGL commands to Python for our use. Now let's explore the program listing.

```
def init():
     glClearColor(0.0, 0.0, 0.0, 1.0)
     gluOrtho2D(-1.0, 1.0, -1.0, 1.0)
```

The first program section is an initialization function.[5] We set the background color to black using the **`glClearColor`** command. This command does not DO anything other than inform OpenGL that when we clear the graphics screen, we are going to use "black" as our background color.[6] The numbers in the parentheses following this command dictate the color and they are, respectively, **`(red, green, blue, alpha)`**. Each color position accepts a value from 0.0 to 1.0 inclusive. You can

[4] Pun intended.
[5] It makes sense to choose variable and function names that describe their purpose. This is a function that sets some initial values, hence the name **`init()`**.
[6] **`glClearColor`** sets the **`GL_COLOR_BUFFER_BIT`** to the color of your choosing.

mix and match colors if you wish and use various intensities.[7] The "alpha" setting deals with the opacity or transparency of the colors. It isn't used as much as the RGB (red, green, and blue) color values and is generally set at 1.0. For example, a purple or magenta background color would be: **`glClearColor(1.0, 0.0, 1.0, 1.0)`**. Note that the mixture of red (1.0) and blue (1.0) will result in a purple/magenta background color.[8] The absence of color (all RGB settings excluding alpha at 0.0) results in black. Setting 1.0 for each color will provide a white background.

The **`gluOrtho2D`** command allows us to set the coordinate system ranges. The command is used this way:

```
gluOrtho2D(x-left, x-right, y-bottom, y-top)
```

In this particular instance, we are setting a coordinate system that ranges from -1.0 to 1.0 in both the ***x*** and **y** axes which places the origin (0,0) in the center of the screen.

Let's look at the **`plotpoints()`** function:

```
def plotpoints():
      glClear(GL_COLOR_BUFFER_BIT)
      glColor3f(1.0, 0.0, 0.0)
```

The **`plotpoints()`** function contains the drawing procedure. Again, the exact function name, **`plotpoints()`**, is not crucial, but should be descriptive of the function's behavior (if possible). The **`glClear(GL_COLOR_BUFFER_BIT)`** command does what you think it should do. It clears the screen (the color buffer bit) and sets the background color to the choice you made in the **`init()`** function. The next line, **`glColor3f(1.0, 0.0, 0.0)`**, sets the plotting color to red. The "**3f**" in the **`glColor3f`** command reminds us that we need to use 3 floating point (decimal between 0.0 and 1.0 inclusive) values for the RGB settings.[9] Placing a 1.0 in the red position and 0.0 in the green and blue positions insures that we will be drawing or plotting using pure red. The **`glColor3f`** command MUST be placed before the plotting or drawing command in order for it to work properly. Take special care that you do not confuse the two OpenGL color commands. **`glClearColor`** is designed specifically to set the background color of the graphics window after it has been cleared. **`glColor3f`** and **`glColor3ub`** are designed to set the color of the plotting pen.

The actual plotting of points occurs in the next program section.

```
glBegin(GL_POINTS)
glVertex2f(0.0, 0.0)
glEnd()

glFlush()
```

[7] Example: **`glClearColor(0.5, 0.8, 0.3, 1.0)`** for an interesting shade of green.
[8] Try it! If the color is too intense, try 0.7 for both red and blue values.
[9] There are other possibilities such as: **`glColor3ub(red, green, blue)`** where "**ub**" stands for "unsigned byte". In this case, the color values range from 0-255.

When we draw or plot in OpenGL, we must inform the graphics system of our intentions. The **glBegin(GL_POINTS)** command serves this purpose. It basically says that we are ready to begin drawing and that we are going to plot points.[10] **glVertex2f(0.0, 0.0)** locates the single point (red, remember?) on the screen using the coordinates (0.0, 0.0), which should represent the origin at the screen's center. The **2f** portion of the **glVertex2f(0.0, 0.0)** command indicates that we are going to use 2 floating point values with **glVertex**, the first to represent the **x** coordinate and the second to represent the **y** coordinate.[11] When we are done plotting, we "pick up our pen" by issuing a **glEnd()** command. Finally, we "flush" our drawing to the screen with **glFlush()**.[12]

In this and subsequent programs we will use a **def main():** function to initiate OpenGL and call any setup routines (such as **init()**) needed by our program. Technically, a separate **main()** function is not required by Python, but the OpenGL tradition is based on the C programming language, which requires a **main()** function. We'll keep with tradition.

```
def main():
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB)
    glutInitWindowSize(500,500)
    glutInitWindowPosition(50,50)
    glutCreateWindow("Plot Points")
    glutDisplayFunc(plotpoints)

    init()
    glutMainLoop()

main()
```

We have seen most of these lines in our first OpenGL program. Note that the graphics window size in **glutInitWindowSize** was set to provide a square drawing surface 600 pixels on a side. The **plotpoints** display function must be properly named and spelled in the **glutDisplayFunc** command.[13] We "call" the **init()** function to set everything up and then put OpenGL/GLUT into an eternal loop to run the program. The final command, **main()**, calls the **main()** function. Without this final command, the program will not run. Try it! Put a **#** in front of **main()** and then run the program.

Now it's time to explore this program further in the exercises!

[10] **GL_POINTS** specifies point plotting. We can draw lines by using **GL_LINES** and triangles by **GL_TRIANGLES**. There are other possibilities for **glBegin** that we will research later.
[11] An ordered pair, just like in algebra!
[12] To this point, the drawing exists only in the computer's memory. We must transfer the drawing from memory to the screen. I'm not certain where "flush" originated in this usage… but we "flush" the memory to the screen using **glFlush()**.
[13] Proper spelling includes making certain that you pay attention to upper and lower case letters! "**plotpoints**" is NOT the same as "**plotpoints**".

Exercises

To avoid overwriting previous exercises, remember to save each new problem as a different program name (such as **ch5ex1.py**)

1) Experiment with the background color settings in **`glClearColor`**. See if you can find both a pleasing and a putrid color. You might want to jot down the settings in your notes or journal for later reference.

2) Change the color of the pixel(s) in the **`glColor3f`** statement. Try plotting a white pixel using (1.0, 1.0, 1.0). Remember to place this command above the **`glVertex2f`** command or it will not work properly!

3) Add the line **`glPointSize(2.0)`** directly above the **`glBegin(GL_POINTS)`** using the same indentation level and see what happens.[14] Experiment with this command by increasing and decreasing the number in parentheses.

4) Plot several points in several sizes and colors. You may insert as many **`glVertex2f`** command as you need between the **`glBegin(GL_POINTS)`** and **`glEnd()`** commands. What happens if you plot a point that is beyond the **`x`** and **`y`** axis ranges set in the **`gluOrtho2D`** command? Try it!

5) Change the **`gluOrtho2D`** command so that the **`x`** and **`y`** axis ranges are from -10.0 to +10.0. Plot some points in those ranges. Can you predict where the points will appear before you run the program?

6) Experiment: Comment out[15] the **`glPointSize(2.0)`** command introduced in exercise 3. Change the **`glBegin(GL_POINTS)`** to **`glBegin(GL_LINES)`** and plot two different points. What happens? See if you can draw a triangle. Uncomment the **`glPointSize`** line. Does anything change? Comment this line again.

7) Experiment: Replace **`GL_LINES`** with **`GL_LINE_STRIP`** and plot the points (0.0, 0.0), (1.0, 1.0), and (-1.0, 1.0). What happens? Now try **`GL_LINE_LOOP`** with the same three points. Was the result the same?

8) Place the command **`glLineWidth(3.0)`** just below the commented **`glPointSize(2.0)`** command. Run the program using any of the **`GL_LINES`**, **`GL_LINE_STRIP`**, or **`GL_LINE_LOOP`** commands.

9) Experiment: Try to draw an equilateral triangle using three different colors, one for each side.

10) Experiment: Try to draw a set of coordinate x and y axes using **`GL_LINES`**.

---

[14] To insert a blank line, you can place the cursor at the end of the line prior to **`glBegin`** and press Enter. You can also place the cursor at the beginning of **`glBegin`** and press Enter.
[15] Using a # at the beginning of a line is a great way to temporarily disable a command.

11) Using the coordinate axes from Exercise 10 as a reference, see if you can plot "mirror" image points simply by changing the signs of your ordered pairs. Here is an example **`def plotpoints():`** function that can serve as a model. Note the use of variables to "store" the value of the ordered pairs. Make certain that you set a background color other than black. Why and where would you do this?

```
def plotpoints():
     glClear(GL_COLOR_BUFFER_BIT)

     # First draw x and y axes
     # Using black as a color
     glColor3f(0.0, 0.0, 0.0)
     glBegin(GL_LINES)
     glVertex2f(-1.0, 0.0)
     glVertex2f(1.0,0.0)
     glVertex2f(0.0, 1.0)
     glVertex2f(0.0, -1.0)
     glEnd()

     # Store an ordered pair in variables
     x = 0.5
     y = 0.5

     # Plot points in bright red
     glColor3f(1.0, 0.0, 0.0)

     # Increase the point size
     glPointSize(3.0)
     glBegin(GL_POINTS)

     # Plot the point
     glVertex2f(x, y)

     # Plot the mirror image or reflection of the point
     # in the x axis.  Note the sign change!
     glVertex2f(x, -y)

     glEnd()
     glFlush()

# End of plotFunc()
```

Figure 5.1 illustrates the simple symmetry from the above code. Note how the points at (0.5, 0.5) and (0.5, -0.5) are reflections of each other across the x axis.

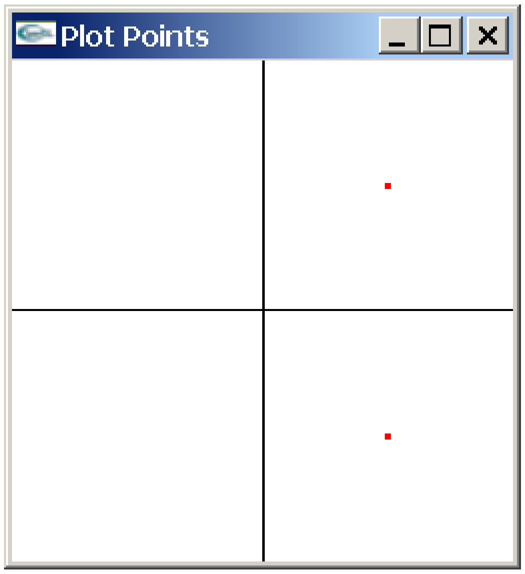


Figure 5.1

Can you reflect across the y axis? How would the signs in the ordered pairs change for a y axis reflection? Can you find a way to reflect across the **`y = x`** diagonal line? You can add a **`y = x`** diagonal by adding the following lines of code within the **`glBegin(GL_LINES)`** section in **`def plotpoints():`**

```
glVertex2f(-1.0,-1.0)
glVertex2f(1.0, 1.0)
```

Here is an example of a symmetry or reflection across the **`y = x`** diagonal.

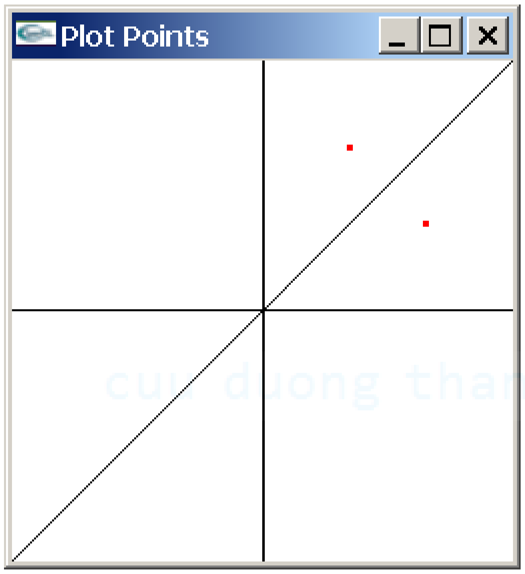


Figure 5.2

Can you reproduce Figure 5.2 yourself?

Now for the challenge. How many other symmetries or mirror images can you create using this idea? Try to predict the outcome before you run the program.

Symmetry is an interesting and valuable concept in math, science, and computer graphics.

12) Expand the concept of point symmetry and see if you can produce a plot illustrating the reflection of a triangle (using **`GL_LINES`**) across the **`x`** axis. If you are successful, try the **`y`** axis as well. Finally, for a challenge, reflect the triangle across the **`y = x`** axis.

13) Create something on your own by plotting various lines and points in different sizes and colors. You may actually create a work of art! Make certain that you THINK about and PREDICT the results before you run the program. This is a scientific method. You establish a thoughtful (theory) prediction (hypothesis) and you test the prediction by running the program (experiment). If the program doesn't behave as you expect, then your hypothesis was incorrect and you think about the problem some more. Eventually, through thought, prediction, and testing, your program will work (probably).

Note: You may have realized this by now, but the most common errors in programming (thus far) involve spelling, capitalization, forgetting to close parentheses. If you generate an error in your program and you can't figure out where the error is in the program statement, look at the line(s) ABOVE the error. It may be that you have forgotten a closing parenthesis somewhere in the line or lines preceding the error.

Section 5.2 Plotting 2D Functions

Let's make Python and OpenGL do something a bit more useful. One of the tasks that most algebra students dislike is graphing functions.[1] With some minor modifications, we can use the program in the last section to create our own function plotter. We'll start with this code:

```
# PyFunc.py
# Plotting functions

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

def init():
     glClearColor(1.0, 1.0, 1.0, 1.0)
     gluOrtho2D(-5.0, 5.0, -5.0, 5.0)

def plotfunc():
     glClear(GL_COLOR_BUFFER_BIT)
     glColor3f(0.0, 0.0, 0.0)
     glPointSize(3.0)

     glBegin(GL_POINTS)
     for x in arange(-5.0, 5.0, 0.1):
          y = x*x
          glVertex2f(x, y)
     glEnd()
     glFlush()

def main():
     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB)
     glutInitWindowPosition(50,50)
     glutInitWindowSize(400,400)
     glutCreateWindow("Function Plotter")
     glutDisplayFunc(plotfunc)

     init()
     glutMainLoop()

main()

# End of program
```

[1] I didn't like it either! I wish I had Python and OpenGL… or even a computer, "way back then".

Save your program and try running it. You should see something like Figure 5.3. We are obviously trying to plot a parabola, but how is our program managing such a feat? First, you should notice that we've added a "**`from Numeric import *`**" to our import statements.[2] This command adds some additional important math statements to Python for our use, as we'll see in a moment. In our program listing, we've changed the **`gluOrtho2D`** command so that the **`x`** and **`y`** ranges are from -5.0 to +5.0 to enlarge the domain and range of the "canvas" on which we are plotting. You may change these values as needed for specific plots.[3] For example, we may need a range of -50.0 to 50.0 to see large scale details in a function or we may need a range of -0.25 to 0.25 to view intricate details of a complex graph near the origin. **`gluOrtho2D`** allows us to employ such domain and range options with great flexibility!

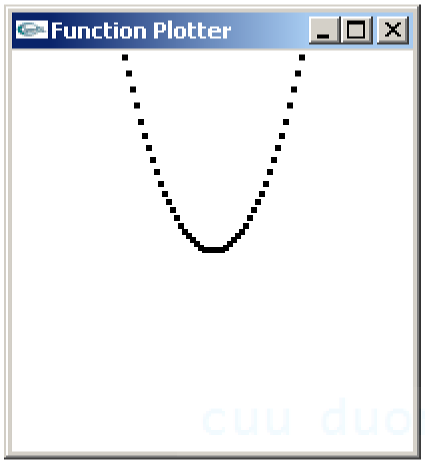


Figure 5.3

The major changes in this program are in the display function **`plotfunc()`**. The background color is white, the plot color is set to black (where and how?), and the size of the points has been increased to 3.0 (what line does this?). The "meat" of the program is found in the following section:

```
glBegin(GL_POINTS)
for x in arange(-5.0, 5.0, 0.1):
      y = x*x
      glVertex2f(x, y)
glEnd()
```

After **`glBegin(GL_POINTS)`**, we see a **`for`** loop similar to the one used in the "Super-3" program. Instead of using **`range`**, though, we use **`arange`** which allows us some freedom to choose how our **`x`** axis values are chosen. The **`range`** command we used previously works only with a list of integers. We need to be able to use decimal numbers in order to create smoother plots. Remember the **`from Numeric import *`** statement? That particular module provides us with the **`arange`** command used in this section of code. The **`for x in arange(-5.0, 5.0, 0.1):`** statement translates to

[2] The newest Python Numeric module is called numpy, so **`from numpy import *`** may be necessary instead of Numeric.
[3] Several exercises will require changes in the domain and range in **`gluOrtho2D`**

"**let x take on all values from -5.0 to +5.0, stepping by 0.1**".[4] The next line, **`y = x*x`** takes each value for **`x`** in the specified range, squares that value, and stores the result in the variable **`y`**.[5] As expected, the **`glVertex2f(x,y)`** command plots both the x and y points for us.[6] As in the previous program, we close **`glBegin()`** with **`glEnd()`** and **`glFlush()`** causes the graphics (heretofore stored only in the computer's memory) to be "flushed" to the screen. The **`def main()`** function containing the GLUT statements is very similar to those in the program in the previous chapter.

Before we do some exercises based on this program example, let's add some coordinate axes for visual reference.[7] Directly below the **`glPointSize(3.0)`** command, add the following lines:

```
glBegin(GL_LINES)
glVertex2f(-5.0, 0.0)
glVertex2f(5.0, 0.0)
glVertex2f(0.0, 5.0)
glVertex2f(0.0, -5.0)
glEnd()
```

Now when the program is executed, we see the graph of the function y = x^2 in a more familiar setting as shown in Figure 5.4.

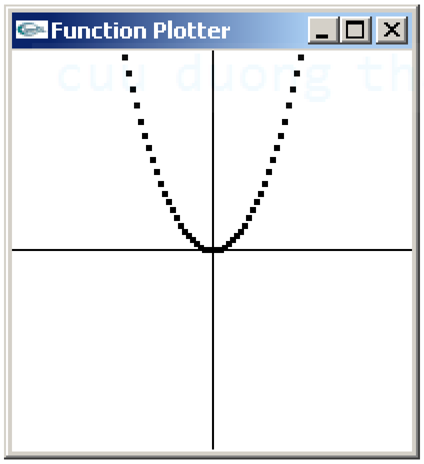


Figure 5.4

The **`GL_LINES`** parameter in the new **`glBegin()`** statement takes pairs of points in **`glVertex2f(x,y)`** format and draws a straight line between those points. In this example, the **`x`**-axis is drawn first from **`glVertex2f(-5.0,0.0)`** to **`glVertex2f(5.0,0.0)`** and the **`y`**-axis is drawn from **`glVertex2f(0.0,5.0)`** to **`glVertex2f(0.0,-5.0)`**. You might try plotting the function using a different color for contrast.

[4] Actually, both the **`range`** and **`arange`** commands stop one step less than the higher number. In this case, 4.9 is the last value for **`x`**. You might try replacing the "**`0.1`**" step with "**`0.01`**"?
[5] You could also type "**`y = x**2`**". Recall how we cubed a number in the "Super-3" program?
[6] Remember that <u>all indented lines</u> after the **`for`** statement are included in the loop.
[7] As in Exercise 10 in the previous section.

Here are some comments on representing functions using computer statements. If we want to graph the function $f(x) = 2x^3 - 3x^2 + x - 5$, we simply can't type the equation into Python as it appears in our textbook. We must translate the function into a form that is understood by Python. Unless you are using a computer algebra system such as Mathematica, all such equations require translation regardless of the computer language you are using. In this case, we translate $f(x) = 2x^3 - 3x^2 + x - 5$ as follows:

```
y = 2*x**3 - 3*x**2 + x - 5
```

This isn't too difficult. Notice that we must specifically tell Python to multiply and raise **`x`** to a power. We can't type **`2x`** and expect Python to understand that this means "**2 times x**". In Python, **`2x`** is written **`2*x`**. As long as you keep in mind that every operation must be specifically typed and that the order of arithmetic must be strictly obeyed, then you shouldn't have too many problems.

One final note: In Python (or in most languages), the "**`=`**" sign has a couple of meanings. First, the "**`=`**" sign may act as an assignment statement. The equation **`x = x + 1`** has a nonsensical meaning in algebra. How can **`x`** equal itself plus one? In Python, this statement is translated: Add one to **`x`** and then assign, store, or place the new value back into **`x`**. In other words, the new value of **`x`** will be the old value of **`x`** plus one. This creates a counting device! So, anytime Python encounters an "**`=`**" sign, it will evaluate the right hand side of the "**`=`**" sign first and then assign that value to the variable on the left side of the "**`=`**" sign. The second use of an "**`=`**" sign is in comparisons such as an **`if`** statement. In this usage, we employ a "**`==`**" sign to distinguish it from a simple assignment statement. As an example, we might say: **`if x == 3:`** meaning that if x equals 3, then we'll do something important.[8] If you forget and use "**`=`**" in an **`if`** statement rather than "**`==`**", the Python interpreter will generate an error message.

Exercises

1) Graph the following equations. Don't forget to translate the equations into Python!

   a) $y = x^2 - 2$
   b) $y = x^3 - 3x - 1$
   c) $y = x^4 - 5x^3 + x^2 - 3x - 1$
   d) $y = \sin(x)$
   e) $y = \sin(3x)$
   f) $y = \sin(x/3)$
   g) $y = \cos(x)$

2) How would you identify the roots (if they exist) of the functions in Exercise 1?

3) Experiment with the **`glPointSize()`** statement. Does this statement apply only to the points plotted or does it apply to the **`x`** and **`y`** axis lines as well?

[8] The idea of exact equality introduces the possibility of some interesting side-effects. Floating point values are represented by binary numbers (base 2) internally, so if we pose the conditional **`if x == 2:`**, which is an integer with an exact binary representation and then check to see **`if x == 2.0:`**, which is a floating point value with a (perhaps) inexact representation, we may not get the expected result. This is only an example, but please keep this idea in mind for later use.

4) Replace **`GL_POINTS`** with **`GL_LINES`** in the **`glBegin()`** statement above the function plot section of the program. What happens? Why does this happen? What about **`glPointSize()`** mentioned in exercise 2?

5) Replace **`GL_LINES`** with **`GL_LINE_STRIP`** in exercise 3. What happens?

6) Experiment with **`GL_LINES`** and **`glLineWidth(2.0)`** in place of **`glPointSize()`**.[9] Try various line widths, including decimal values.

7) Change **`GL_LINES`** back to **`GL_POINTS`**. Experiment with the **`arange`** command. Change the **`arange(-5.0, 5.0, 0.1)`** to a **`arange(-10.0,10.0, 0.01).`** You may have to edit the **`gluOrtho2D`** command in the **`init()`** function for this change to display properly. Now try a step (in **`arange`**) of **`0.001`** followed by a step of **`0.0001`**. The smaller the step, the nicer the plot… but it can take a bit longer to draw because you are plotting more points. You can also modify the step size while using **`GL_LINES`** for a smoother or coarser solid curve.

8) See if you can plot multiple functions in the same graphics window. You might consider something like this in **`def plotfunc():`**

```
glBegin(GL_POINTS)
for x in arange(-5.0 ,5.0, 0.01):
     y = x*x
     a = x + 1
     glVertex2f(x, y)
     glVertex2f(x, a)
glEnd()
```

which should give you a result something like Figure 5.5. Try some other functions and see where points of intersection occur.

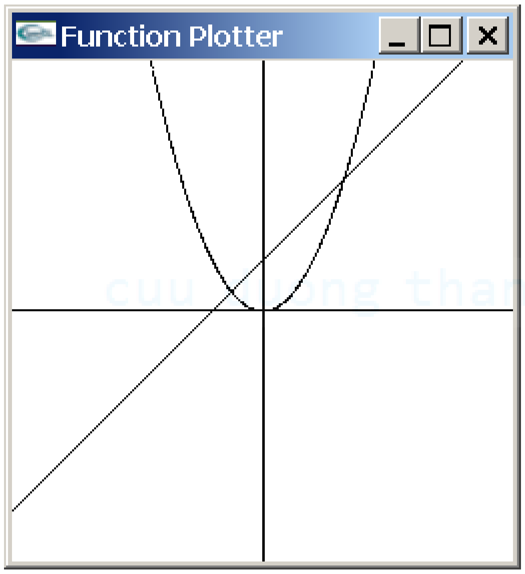


Figure 5.5

[9] Remember, **`glLineWidth`** only works with lines. **`glPointSize`** works only with points!

9) Plot **`y = sin(x)`** and **`y = cos(x)`** on the same graph. Use different colors for each function and also include coordinate axes for reference. What do you notice about the two function plots? How are sin and cos related? How are they different?

10) Building on Exercise 8, can you draw a circle? The equation for a circle is:

$x^2 + y^2 = r^2$ or **`y = sqrt(r*r – x*x)`** or **`y = sqrt(r**2 – x**2)`**

You can't enter the traditional implicit equation for a circle as shown in the first equation above. You MUST use the second or third explicit form, **`y = sqrt(r*r – x*x) or y = sqrt(r**2 – x**2).`** But how would you go about accomplishing the feat of drawing a complete circle? A circle is NOT a function according to the vertical line graph test in algebra. Think about this and find a solution. Hint:

```
r = 1.0
glBegin(GL_POINTS)
for x in arange(-1.0, 1.0, 0.01):
     y = sqrt(r**2 – x**2)
     glVertex2f(x, y)
     # do we need another glVertex2f statement here?
glEnd()
```

Notice that I've added an **`r = 1.0`** statement above the **`glBegin`**. Why? Also notice that I've changed the **`for`** loop from:

```
for x in arange(-5.0, 5.0, 0.01):
```

to:

```
for x in arange(-1.0, 1.0, 0.01):
```

Again, why? Try the original **`for`** loop and see if you can figure out the error.

11) At some point in algebra we find ourselves graphing inequalities such as $y < x^2$. How could we do this using Python? First, plot the graph of $y = x^2$. Use **`glPointSize(1.0)`** and a **`for`** loop such as:

```
for x in arange(-5.0, 5.0, 0.01):
```

Next, modify the **`plotfunc`** display function as follows:

```
def plotfunc():
  glClear(GL_COLOR_BUFFER_BIT)
  glColor3f(0.0, 0.0, 0.0)
  glPointSize(1.0)

  glBegin(GL_POINTS)
  for x in arange(-5.0, 5.0, 0.01):
          y = x*x
          glColor3f(0.0, 0.0, 0.0)
          glVertex2f(x,y)
          for a in arange(-5.0, 5.0, 0.01):
               if a < x*x:
                    glColor3f(0.50,0.50,0.50)
                    glVertex2f(x,a)
     glEnd()

     glColor3f(0.0, 0.0, 0.0)
     glBegin(GL_LINES)
     glVertex2f(-5.0, 0.0)
     glVertex2f(5.0, 0.0)
     glVertex2f(0.0, 5.0)
     glVertex2f(0.0, -5.0)
     glEnd()

     glFlush()

# End of plotfunc block
```

You should also change **gluOrtho2D** as follows if you haven't done so already:

```
     gluOrtho2D(-5.0, 5.0, -5.0, 5.0)
```

The shaded region of the inequality is visible in Figure 5.6

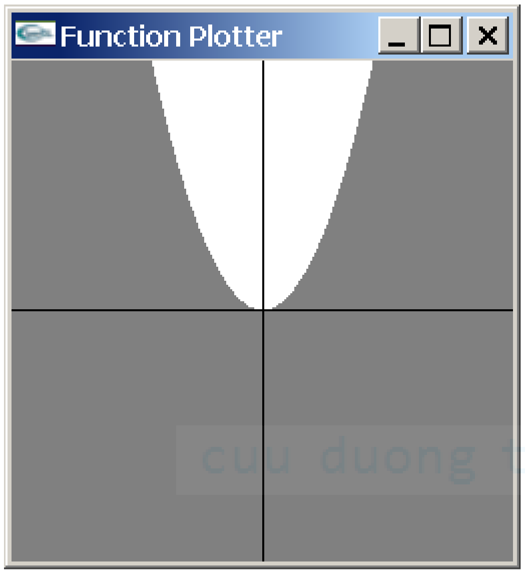


Figure 5.6

Notice the outline of the original y = x^2 function is still visible. Also notice the addition of a nested loop containing the variable **a**.[10] It is this nested loop that is creating

[10] Think of **a** in the same terms as **y**. In the nested loop, we are calculating all values of **a** and coloring them ONLY if they are less than **x*x** or x^2.

the shaded region. Can you shade the area above the parabola? Try it! Now here's the challenge. Using this idea, see if you can revisit exercise 8 and Figure 5.5 and shade the region bounded by the line, the parabola, and the constraint that both x and y must be greater than 0. The solution should look like Figure 5.8. Here is a hint: The line **`if a < x*x and x > 0 and a > 0:`** when properly used will produce a plot that looks like Figure 5.7. See if you can build on this concept and reproduce Figure 5.8.

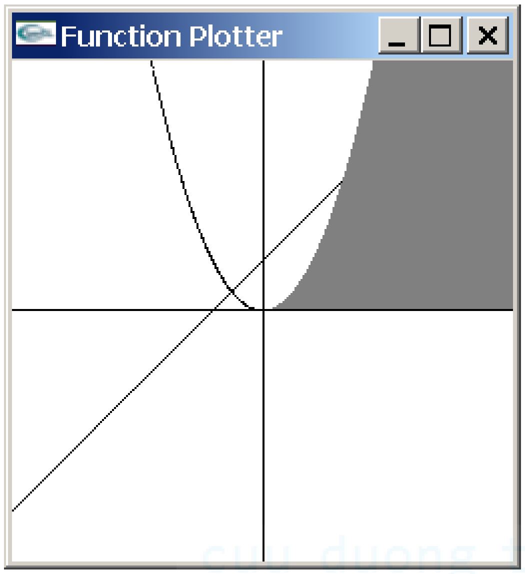


Figure 5.7

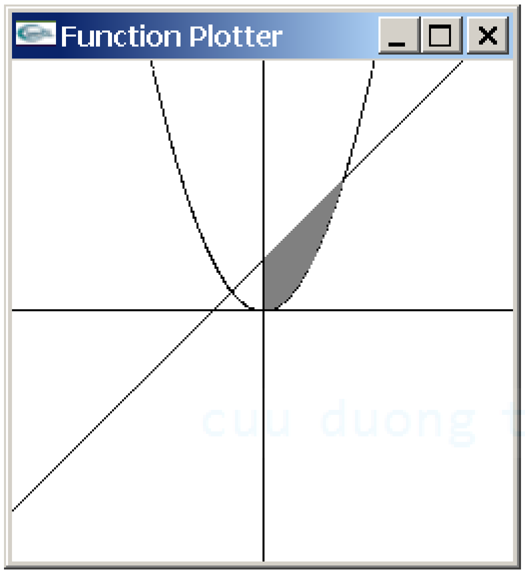


Figure 5.8

<u>Note</u>: Ok, by this time you should be making an attempt to predict the output of your program prior to running it for the first time. What do you expect the program to do? If the program doesn't run, then what is wrong with your code? Fix the bugs! If it runs, but doesn't do what you predict, then rethink your prediction. This is very much a scientific method! you should also be experimenting on your own in an attempt to expand on the exercises and concepts presented in the chapter. Only in this manner will you truly master the material!

You should also be experimenting on your own in an attempt to expand on the exercises and concepts presented in the chapter. Only in this manner will you truly master the material!

Sections 5.3 Parametric Equations

Functions are defined in such a fashion that for each value of **x**, there can be one and only one value of **f(x)**.[1] This means that we can't easily plot circles unless we plot them piecewise by first plotting the top half of the circle followed by the bottom half in another section of code. By using parametric equations, we can "fix" this problem. Parametric equations behave much like an "Etch-a-sketch". The **x** (horizontal control) and **y** (vertical control) are separately manipulated and the result can be a fascinating curve with none of the "vertical line test" limitations. In order to create a parametric system of equations, we define both the **x** and **y** equations in terms of another variable such as **t**. Examples of parametric equations are as follows:

x = sin(t)
y = cos(t)

The equations that define **x** and **y** are independent of each other (like the knobs on an "Etch-a-Sketch"), but together they define a single curve. Let's see how this works. <u>Using the program we created in the last section</u>, let's modify the **`def plotfunc():`** function to accept parametric equations. First, make certain the **`gluOrtho2D`** command in the def init() function has **x** and **y** ranges of **`-2.0`** to **`2.0`** respectively. Then change the **`def plotfunc():`** to the following:

```
def plotfunc():
     glClear(GL_COLOR_BUFFER_BIT)
     glColor3f(0.0, 0.0, 0.0)
     glPointSize(1.0)

     # Plot the coordinate axes
     glBegin(GL_LINES)
     glVertex2f(-2.0, 0.0)
     glVertex2f(2.0, 0.0)
     glVertex2f(0.0, 2.0)
     glVertex2f(0.0, -2.0)
     glEnd()

     # Plot the parametric equations
     glBegin(GL_POINTS)
     for t in arange(0.0,6.28, 0.001):
          x = sin(t)
          y = cos(t)
          glVertex2f(x, y)
     glEnd()
     glFlush()

# End plotfunc()
```

[1] y = f(x) where f(x) is a common designation for a function of the variable x. Functions must pass the "vertical line test", meaning that if we pass a vertical line through any point on the function graph, it can never intersect the graph in more than one point.

Remember that in Python you MUST preserve the indentation scheme as shown in the code listing above in order for the program to work properly.

Save the program as **PyParam.py** or something similar. You may want to change the comment statements at the beginning of the program to reflect the new program title and program operation. If you run the program, you should see something like Figure 5.9. It's a circle![2] Now the fun begins! What happens if we change the equations for **x** and **y**? What happens if we let **t** range from **-6.28** to **6.28**?[3] Try it! Change the **x** and **y** equations to: `x = cos(3*t)` and `y = sin(5*t)` respectively and change the range of **t** to: `for t in arange(-6.28, 6.28, 0.01):`

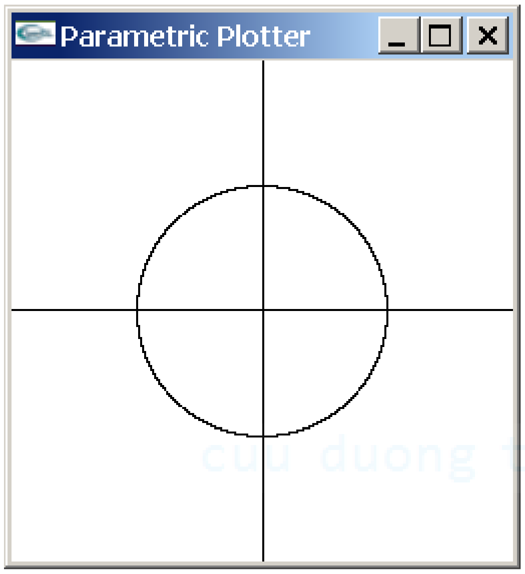


Figure 5.9

If you followed the directions on the previous page, your next plot should like something like Figure 5.10 on the next page. I took the liberty of remarking out the lines that draw the **x** and **y** axis. Isn't this neat?[4]

You may be wondering how the circle was plotted in Figure 5.9 or how the neat Lissajous curve was drawn in Figure 5.10? In order to understand these and other remarkable plots, we need to briefly discuss the basic trigonometry functions and how they work to produce these fascinating drawings. You should use Figure 5.11 on the next page as a reference.

[2] 6.28 is approximately 2π. There are 2π radians in a circle. See footnote 3.
[3] Of course, these are approximations to -2π and 2π just as the original range of **t** was from 0 to 2π. Trig functions in Python and other programming languages are based on radians rather than degrees (2π radians = 360^o).
[4] Research Lissajous or Bowditch curves online. They are remarkable!

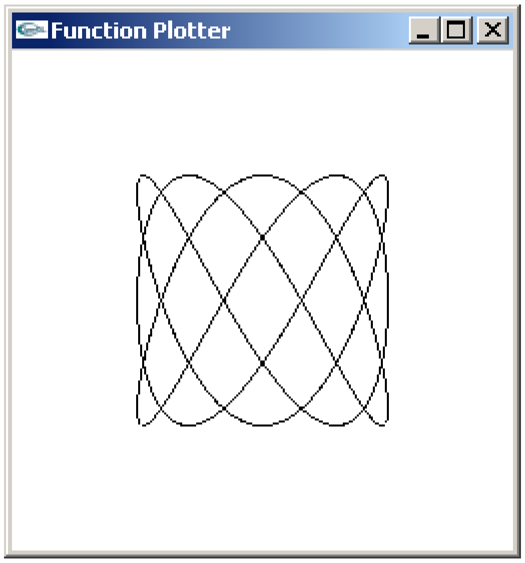


Figure 5.10

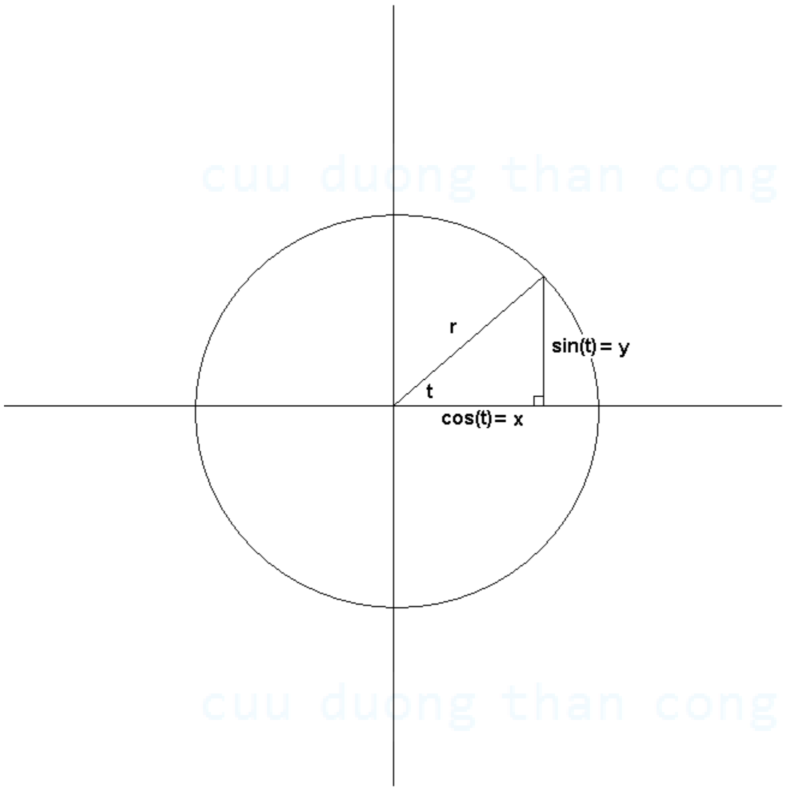


Figure 5.11

You may (or may not) recall the definitions of the trig functions. The **sin** of angle **t** is defined as the side opposite angle **t** (in this case the vertical "**y**" axis leg in the right triangle in Figure 5.11) divided by the hypotenuse **r**. Therefore, sin(t) = $\frac{y}{r}$. We normally assume that r = 1,[5] and then **y = sin(t)**. So, the value of **sin(t)** controls the "**y**" knob on our computer "Etch-a-Sketch". Likewise, the **cos** of angle **t** is defined as the side adjacent to angle **t** (in this case the horizontal "**x**" axis leg in the right triangle in Figure 5.11) divided by the hypotenuse **r**. So, cos(t) = $\frac{x}{r}$ and if r = 1, **x = cos(t)**.[6] As before, the value of **cos(t)** controls the "**x**" knob on our computer "Etch-a-Sketch". If the value of **r** is held constant (r = 1 or some other value), then as **t** "sweeps around" like a radar scope, we draw a perfect circle! Modifying **t** inside the trig functions "twiddles the knobs" and we can get some very interesting graphics as seen previously in Figure 5.10.

The exercises will allow you to explore these concepts further. Save a copy of the original program for this section as a reference.[7] Have fun!

Exercises

Note: Be prepared to make modifications to the original program as specified by the exercises. Most of the modifications are minor, but it's important to retain the original version of the program in this section for future reference. Why? So you can save time! Later we will develop a skeleton program that will help save you quite a bit of typing. At the moment our programs are not lengthy, but by saving the original program you can save some time by not having to retype every line. Each new program you create in any exercise should be saved under a different name such as **ch5ex3.py** or something equally meaningful.

1) The equations in the program that determine the values of **x** and **y** are:

```
x = cos(t)
y = sin(t)
```

You can alter these equations by changing them to:

```
x = (c*t+d)*sin(t)
y = sin(a*t+b)
```

and adding the lines:

```
a = 0.5
b = 0.5
c = 0.25
d = 0.0
```

[5] This makes everything nice and easy. Actually, **r** can be any value we choose and everything still works just fine. A small circle has the same angles and trig values as a large one… think of a bulls-eye target. If you sweep through 360^o or 2π radians on the largest circle, you do the same for the smallest. So why not choose **r = 1**?

[6] The **tan(t)** is the opposite side divided by the adjacent side or $\frac{\sin(t)}{\cos(t)}$.

[7] Any new exercises should be named… and saved… as something different than the original program. That way you will always have easy access to the original code.

just above the **`glBegin(GL_POINTS)`** statement to define the variables.[8] Experiment with different values of **`a`**, **`b`**, **`c`**, and **`d`**. If the plot goes beyond the graphics window border, you may increase the **`x`** and **`y`** axis ranges in the **`glOrtho2D`** statement. Be prepared to change these values back to their original state if needed in future plots.

2) Try using **`glBegin(GL_LINES)`** instead of **`glBegin(GL_POINTS)`** in the previous and future programs. Also, try different ranges for **`t`** in the **`for`** loop. You may find that if the range is large enough, the plot doesn't change. What this means is that the plot is retracing itself as **`sin(t)`** and **`cos(t)`** revisit the same values.[9]

3) Try plotting an ellipse using the following parametric equations.[10]

```
x = a*cos(t)
y = b*sin(t)
```

Try values of **`a = 1.5`** and **`b = 0.50`** (as in exercise 1, define the variables BEFORE you use them) and see what happens. You should see something like Figure 5.12 on the next page. The "long" axis is called the major axis of the ellipse and the short axis is called the minor axis. How could you change the values for parameters **`a`** and **`b`** so that the major axis was vertically oriented rather than horizontally oriented?

What happens if you type in the same values for **`a`** and **`b`**? Try **`a = 1.5`** and **`b = 1.5`**, then try **`a = 1.0`** and **`b = 1.0`**. Does the resulting plot make sense based on the values you typed? How does this compare to the original parametric equations for a circle at the beginning of this section?

So, do you think that ellipses and circles are related? Could you say that a circle was simply a special case of an ellipse… one in which both major and minor axes are equal?

4) Change the equations to:

```
x = sin(t)
y = sin(a*t + b) + c*sin(d*t + e)
```

and add a line: **`e = 6.0`** below the other variable assignments. Try values of **`a = 2.0, b = 1.0, c = 1.5, d = 3.5`**, and **`e = 6.0`** and then run the program. Experiment with other values for these variables. Change the for loop to:

```
for t in arange(-6.28, 6.28, 0.001):
```

and set the axis ranges as follows:

[8] Such variables are often called parameters.
[9] Sin and Cos take on the values from -1.0 to +1.0 inclusive. You might try plotting a **`sin(t)`** or **`cos(t)`** function (or both) using what you learned in section 5.2. You will see how both functions continually revisit all values between -1.0 and +1.0 as the angle size increases without bound.
[10] Remember to replace or comment out the "old" **`x`** and **`y`** equations.

```
gluOrtho2D(-3.0, 3.0, -3.0, 3.0)
```

If you are also stilling plotting **x** and **y** axis lines, you should change those statements to reflect the new ranges above. How?

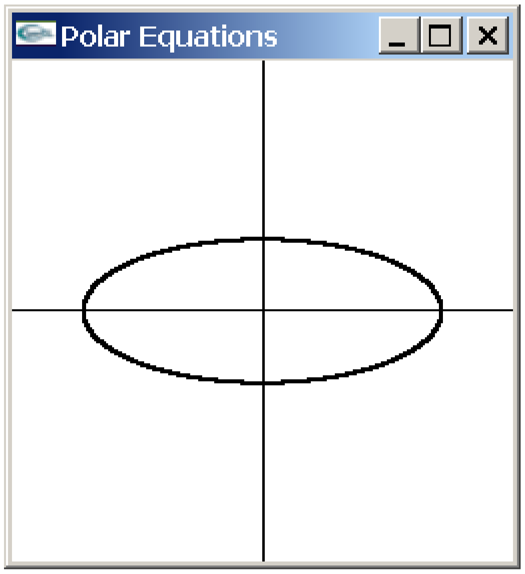


Figure 5.12

5) Experiment with the range and step in **arange** until you notice changes in the program. For example, you might try: **for t in arange(-1.0, 1.0, 0.1):** and then increase the range and decrease the step until the graph remains unchanged. Trigonometric functions are periodic, which means that the values generated by such functions have the potential to repeat themselves at certain intervals (periodically). So, do not be surprised if your graphs do not always change when you change loop parameters.

6) First, save the parametric plot program with a new name so that you can preserve the original **def plotfunc():** as a starting point for further experiments. Then, comment out the lines that assign values to each of the variables **a** through **e** and modify **def plotfunc():** so that it looks like the following. Remember to indent at the same level after each **for** statement!

```
def plotfunc():
     glClear(GL_COLOR_BUFFER_BIT)
     glColor3f(0.0, 0.0, 0.0)
     glPointSize(1.0)

     glBegin(GL_POINTS)
     for a in arange(0.1, 2.0, 0.1):
          for t in arange(-4.4, 4.4, 0.01):
               x = 0.3*a*(t*t – 3)
               y = 0.1*a*t*(t*t – 3)
               glVertex2f(x, y)
     glEnd()
     glFlush()
```

```
    # End plotfunc()
```

This code construction is called a nested loop. There are two loops involved; the second **for** loop is inside or "nested" within the first. As the variable **a** in the outer loop takes on each value in the range from 0.1 to 2.0, stepping by 0.1, the inner **t** loop makes a complete cycle through its entire range. This set of equations draws a series of nested figures called Tschirnhausen's Cubic.[11] One entire looping curve is drawn by the **t** loop for each value of **a** produced by the outer loop. Figure 5.13 illustrates Tschirnhausen's Cubic.[12] In order for your plot to look exactly like Figure 5.13, you will need to make certain the **gluOrtho2D** ranges are as follows:

```
gluOrtho2D(-2.0, 2.0, -2.0, 2.0)
```

This is not strictly necessary, however. The plot looks fine with the larger ranges from previous exercises.

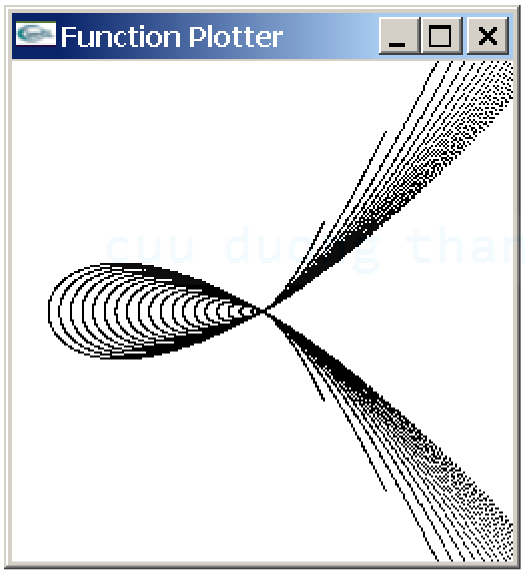


Figure 5.13

7) The next experiment is called Miller's Madness.[13] Change the equations in the **def plotfunc():** in the **original** parametric plotting program[14] at the beginning of this section to:

```
x = sin(0.99*t) - 0.7*cos(3.01*t)
y = cos(1.01*t) + 0.1*sin(15.03*t)
```

and the loop statement to:

```
for t in arange(-200.0, 200.0, 0.005):
```

[11] Dewdney, A. K. (1990). "The Magic Machine". P. 272
[12] Parametric equations don't have to use trig functions as this problem demonstrates.
[13] Dewdney, A. K. (1990). "The Magic Machine". P. 276
[14] You did save it, didn't you?

then save the program with a new name[15] and run it. It may take a few seconds to complete the drawing, but I think it's well worth it as Figure 5.14 shows! I do recommend that you set the x and y axis ranges in **gluOrtho2D** to:

```
gluOrtho2D(-2.0, 2.0, -2.0, 2.0)
```

if you haven't already done so for the most pleasing plot dimensions. You may be thinking at this point that changing the **gluOrtho2D** plot ranges is a method for zooming into a plot. You are correct! However, this is very inefficient. Later on in the text we'll learn how to zoom a bit more efficiently.

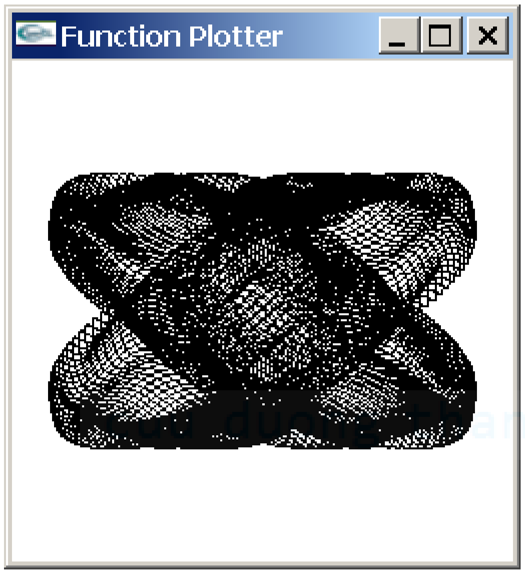


Figure 5.14

8) Invent your own equations for **x** and **y**! If you get something particularly pleasing, make certain you save that program. Also, experiment with different colors for the backgrounds[16] and plot points.

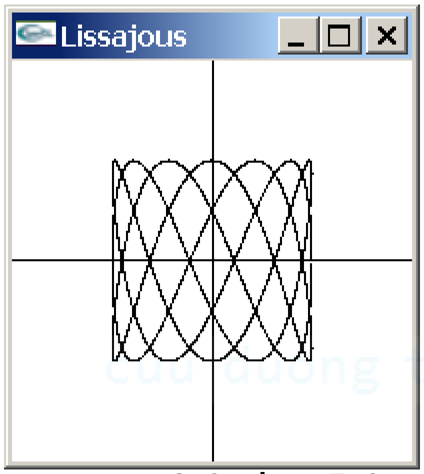
9) This is a challenge exercise. See if you can draw an ellipse that is oriented on a different axis (such as the **y = x** diagonal) rather than the **x** or **y** axis. If you want to do this the easy way, research the **glRotatef** command in the RedBook or online. For a greater challenge, try to accomplish this task using parametric equations. Yes, you may research this topic online, but try to experiment on your own first.

10) Figure 5.10 illustrated a Lissajous/Bowditch curve using **x = cos(3.0*t)** and **y = sin(5.0*t)** as equations. Change these to **x = cos(a*t)** and **y = sin(b*t)** and make certain that you have program lines defining **a** and **b** just above **glBegin(GL_POINTS)** as in exercise 1. Make certain that **t** ranges from 0.0 to 6.28 in the **for** loop with a step of **0.001**. Try the following parameters:

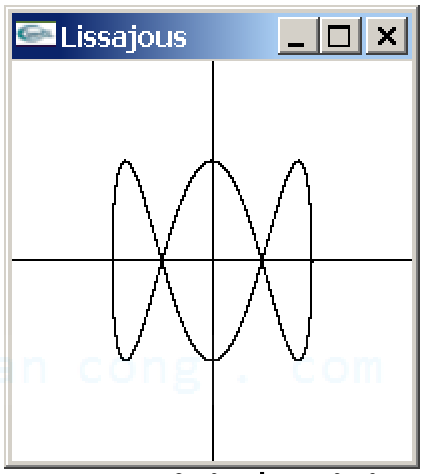
[15] I would suggest **pyMiller.py** or **ch53ex7.py**. You may think you are accumulating a lot of programs, but we've only just begun.
[16] **glClearColor(0.35, 0.79, 0.60)** for an odd shade of turquoise, etc.

`a = 3.0` and **`b = 5.0`** (which should reproduce figure 5.10).
`a = 3.0` and **`b = 7.0`**
`a = 3.0` and **`b = 9.0`**
`a = 3.0` and **`b = 11.0`**
`a = 5.0` and **`b = 11.0`**

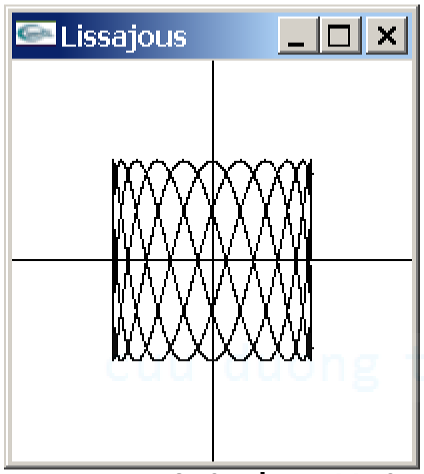
Do you notice a pattern between the values for **a** and **b** and the resulting graphic? Look at the figures below, each labeled with the corresponding values for **a** and **b**. What happens if **`a = b`**? Why does **`a = 3.0`** and **`b = 9.0`** behave as it does? Try some addition combinations and see what happens. What if **a** > **b**, such as **`a = 5.0`** and **`b = 3.0`**? What do the numbers 3, 5, 7, and 11 have in common?



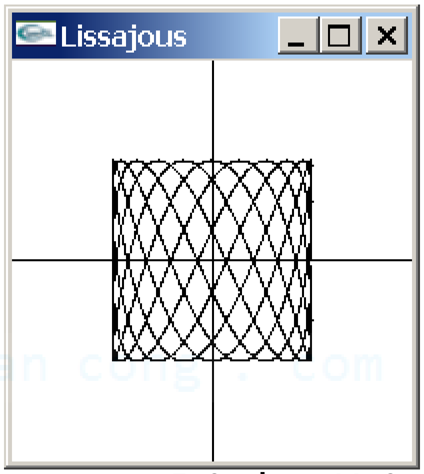
`a = 3.0, b = 7.0`



`a = 3.0, b = 9.0`



`a = 3.0, b = 11.0`



`a = 5.0, b = 11.0`

Note: The following exercises are designed to illustrate some well-known mathematical curves. The figures associated with these curves are found at the end of the exercises.

11) The Astroid curve (no, not an asteroid). Use the following equations for **`x`** and **`y`**.

```
x = a*cos(t)**3
y = a*sin(t)**3
```

Try an **`x`** and **`y`** axis range of -5.0 to 5.0.[17] Did you get an Astroid? What do you think the parameter "a" specifies? Try **`a = 5.0`** and then **`a = 2.5`**. What does the parameter "**a**" do? This curve is also called the tetracuspid because it has 4 cusps.[18] The curve can be formed by rolling a circle of radius **`a/4`** on the inside of a circle of radius **a**.[19] What happens if you modify the **`for`** loop to **`for t in arange(0.0, 3.14, 0.001):`** instead of **`for t in arange(0.0, 6.28, 0.001):`**? Remember there are 2π radians in a complete circle, so **`0.0`** to **`6.28`** is equivalent to 2π.

12) The Cardiod.[20] The Cardiod is drawn by tracing a point on a circle as it rolls around another circle of the same radius. A Cardiod is drawn using the following equations:

```
x = a*(2.0*cos(t) - cos(2.0*t))
y = a*(2.0*sin(t) – sin(2.0*t))
```

Set **`a = 0.5`** or change the **`x`** and **`y`** axis ranges from -2.0 to 2.0. The graphic should look like the Cardiod plot at the end of the exercises. We'll revisit the Cardiod as a polar equation in section 5.5. Again, what purpose does the **a** parameter fulfill in this equation? At times, parameters determine the scale or size of the plot and at other times the parameter may determine the number of some feature of the graph. Feel free to experiment with any and all parameters! Where do you think the name "Cardiod" came from?

Also, feel free to experiment with various colors, point sizes, and line widths! The displayed figures in exercises 10, 11, and 12 were drawn with **`glPointSize(2.0)`**.

13) The Epicycloid.[21] This curve is traced by a point P on a circle of radius **b** which rolls around a fixed circle of radius **a**. Use the following parametric equations:

```
x = (a + b)*cos(t) – b*cos((a/b + 1.0)*t)
y = (a + b)*sin(t) – b*sin((a/b + 1.0)*t)
```

We need to expand the range of the **`x`** and **`y`** axes from -20.0 to 20.0 to see this plot properly. The loop statement should be **`for t in arange(-12.56, 12.56, 0.001):`** to both increase the range of **`t`** and decrease the step size. Try parameter values **`a = 12.0`** and **`b = 2.25`**. Also, modify **`glPointSize`** to **`glPointSize(1.0)`** if necessary.

[17] Modify the **`gluOrtho2D`** statement in the **`init`** function. Figure it out!
[18] The tetracuspid belongs to a family of curves known as the hypocycloids.
[19] Kokoska, Stephen. "Fifty Famous Curves, Lots of Calculus Questions, And a Few Answers". Dept. of Mathematics, Computer Science, and Statistics, Bloomsburg University.
[20] Ibid
[21] Ibid.

Experiment with various ranges of **`t`** and other parameter values for **`a`** and **`b`**. You may need to change the ranges for the **`x`** and **`y`** axes in **`gluOrtho2D`** as you experiment in order to obtain the most pleasing plot. Also be prepared to change the step size in the loop statement if needed. You may find out that a step of **`0.01`** or **`0.001`** is not small enough to obtain an unbroken plot.

14) The Epitrochoid.[22] The circle of radius **`b`** rolls on the outside of the circle of radius **`a`**. The point P is at a distance **`c`** from the center of the circle of radius **`b`**. The parametric equations for the Epitrochoid are:

```
x = (a + b)*cos(t) – c*cos((a/b + 1.0)*t)
y = (a + b)*sin(t) – c*sin((a/b + 1.0)*t)
```

Keep all ranges (x and y axes and the loop range for t) the same as in the previous exercise. Try values of: **`a = 12.0`**, **`b = 2.25`**, and **`c = 5.0`**. Keep **`glPointSize(1.0)`** unless you prefer a “thicker” plot.

Again, feel free to experiment with the parameters **`a`**, **`b`**, and **`c`**. See if you can figure out the purpose of each of the parameters based on your experiments and on the definition of the Epitrochoid.

15) The Hypocycloid.[23] A circle of radius **`b`** rolls on the inside of a circle of radius **`a`**. The point P is on the edge of the circle of radius **`b`**. Try the following equations:

```
x = (a - b)*cos(t) + b*cos((a/b – 1.0)*t)
y = (a - b)*sin(t) – b*sin((a/b – 1.0)*t)
```

Keep the same values for a and b as we started with in the previous two exercises: **`a = 12.0`**, **`b = 2.25`**. Change the **`x`** and **`y`** axis ranges to **`-15.0`** to **`15.0`** by changing the **`gluOrtho2D`** statement in the **`init`** function to **`gluOrtho2D(-15.0, 15.0, -15.0, 15.0)`**.

16) The Hypotrochoid.[24] A circle of radius **`b`** rolls on the inside of a circle of radius **`a`**. The point P is at distance **`c`** from the center of the circle of radius **`b`**. Try the following equations:

```
x = (a - b)*cos(t) + c*cos((a/b – 1.0)*t)
y = (a - b)*sin(t) – c*sin((a/b – 1.0)*t)
```

Keep the **`x`** and **`y`** axis ranges at **`-15.0`** to **`15.0`** as in the previous exercise and the parameters **`a = 12.0`**, **`b = 2.25`**, and **`c = 5.0`**.

17) The Involute of a Circle.[25] The Involute of a Circle is the path traced out by a point on a straight line that rolls around a circle. The equations are:

[22] Ibid.
[23] Ibid.
[24] Ibid.
[25] Ibid.

```
x = a*(cos(t) + t*sin(t))
y = a*(sin(t) – t*cos(t))
```

The **x** and **y** axes should range from **-25.0** to **25.0**. The **t** loop should be:

```
for t in arange(0.0, 25.12, 0.001):
```

Let parameter **a = 1.0**. Your plot should resemble a spiral.

18) The Nephroid.[26] The name Nephroid means kidney-shaped. It is formed by a circle of radius **a** rolling externally on a fixed circle of radius **2a**. Use the following equations:

```
x = a*(3.0*cos(t) – cos(3.0*t))
y = a*(3.0*sin(t) – sin(3.0*t))
```

The **x** and **y** axis ranges should be from **-5.0** to **5.0**. Parameter **a = 1.0**. In the last two exercises, parameters **b** and **c** are not used and may be remarked out or assigned any value ("**1.0**" is preferred). Your Nephroid plot should look like the figure at the end of these exercises (you may have to use your imagination to see a kidney?).

19) Talbot's Curve.[27] There are several forms to this curve, one of which looks like a football. See if you can find the parameters for the football! The equations are:

```
x = ((a**2 + c*c*sin(t)**2)*cos(t))/a
y = ((a**2 – 2.0*c**2 + c*c*sin(t)**2)*sin(t))/b
```

Try the following **for** loop:

```
for t in arange(0.0, 6.28, 0.001):
```

Parameters **a**, **b**, and **c** should be: **a = 1.925**, **b = 4.0**, and **c = 1.725**. The **x** and **y** axis ranges can start out at -2.0 to 2.0, but may need to be expanded for different parameters of **a**, **b**, and **c**.

20) The Triscuspoid.[28] This curve is created by the following equations:

```
x = a*(2.0*cos(t) + cos(2.0*t))
y = a*(2.0*sin(t) – sin(2.0*t))
```

The plot range for both the **x** and **y** axes should be -5.0 to 5.0 and assign parameter **a** the value 1.5 (**a = 1.5)**.

21) The final parametric curve will be the famous "Witch of Agnesi".[29] The equations are simple:

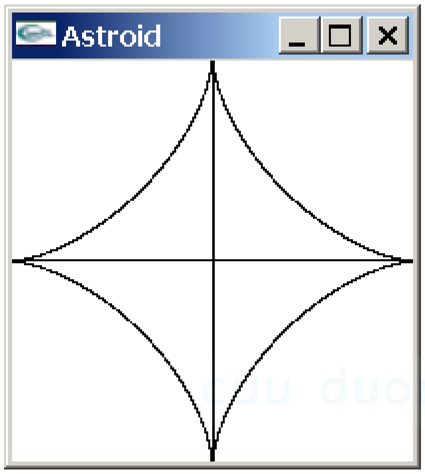
[26] Ibid.
[27] Ibid.
[28] Ibid.
[29] Ibid.

```
x = a*t
y = a/(1.0 + t**2)
```

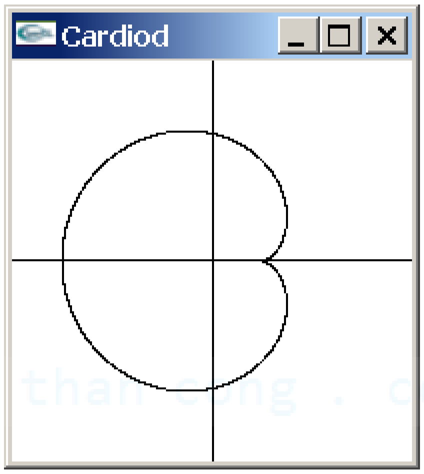
You can set **`a = 1.0`**. The **`x`** and **`y`** axis plot range should go from -2.0 to 2.0 and the loop should be:

```
for t in arange(-2.0, 2.0, 0.001):
```

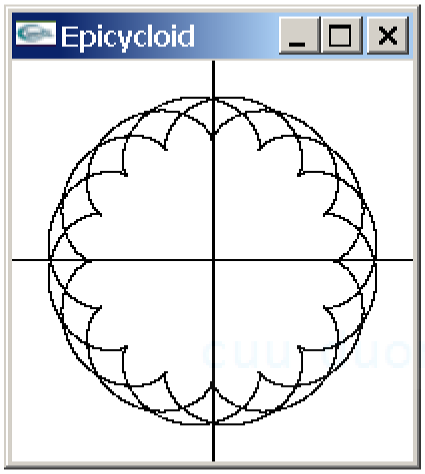
Note that we are not using trig functions in this plot, so we don't have to concern ourselves with multiples of π radians. The plot is not very scary for a "witch". Research this curve and find out how it got its name.[30]



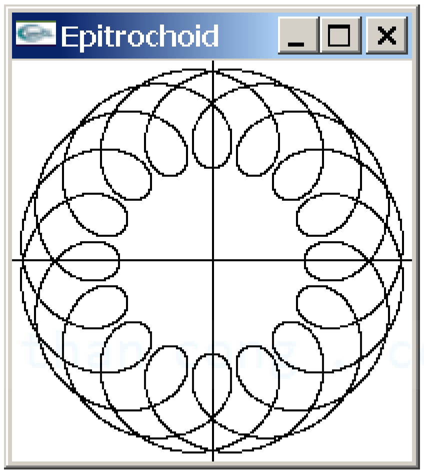
Astroid



Cardiod

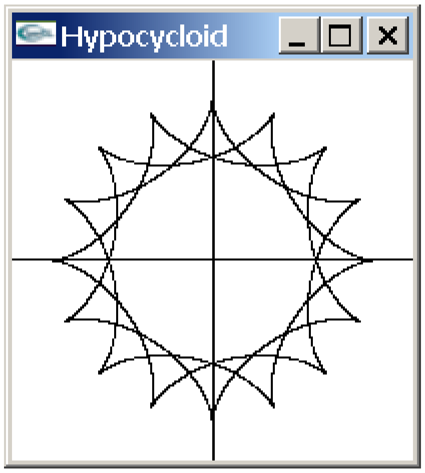


Epicycloid

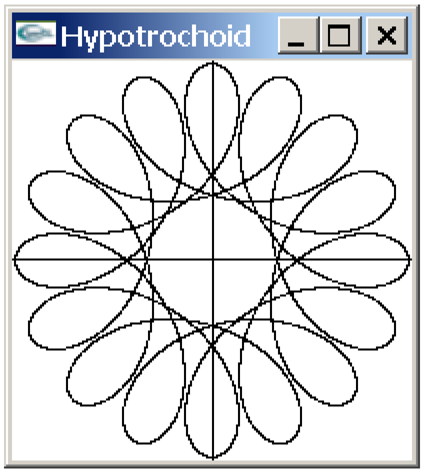


Epitrochoid

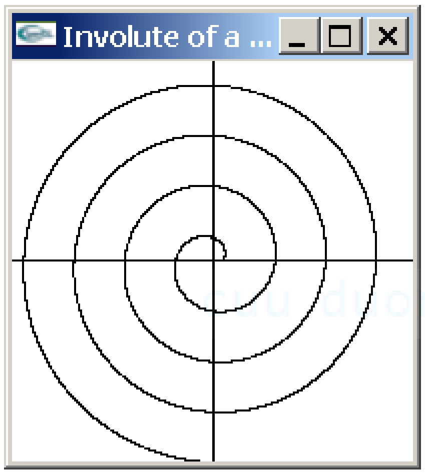
[30] I have been a bit remiss in using the word "research". My intent is for you to find at least 3 different resources online (or in a text or journal) that are consistent in their definitions. In this new world of internet information, one must be <u>VERY</u> careful in choosing a source upon which to base a conclusion. Don't believe everything you read or see online!



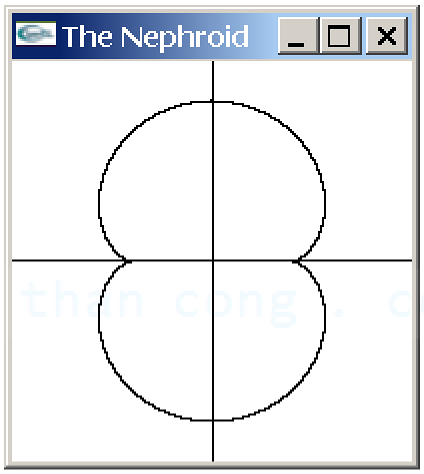
Hypocycloid



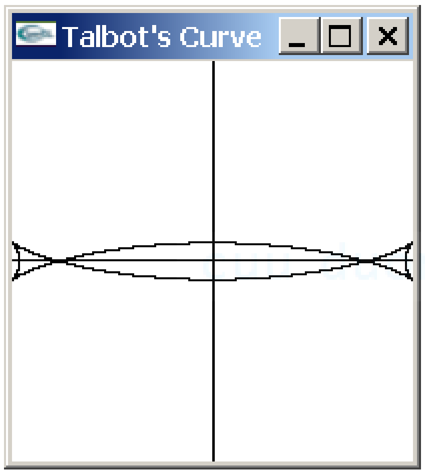
Hypotrochoid



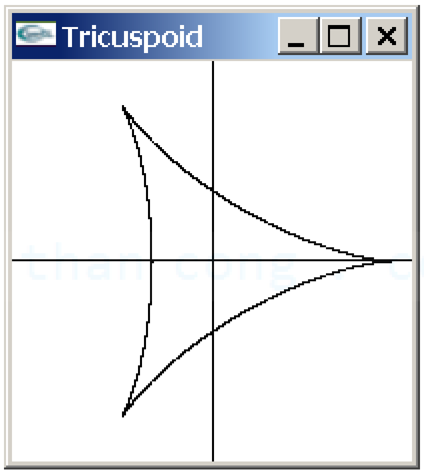
Involute of a Circle



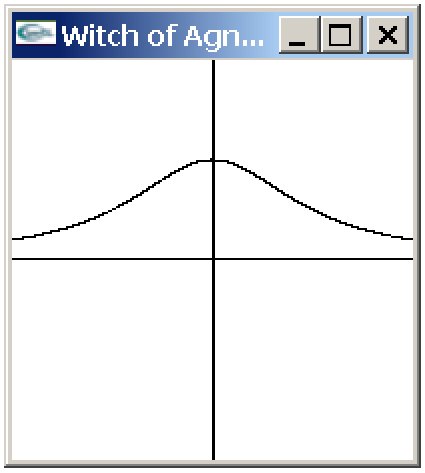
The Nephroid



Talbot’s Curve



Tricuspoid



Witch of Agnesi

Section 5.4 An Example from Physics

One of the first uses (maybe even the first use?) of computers was to calculate the trajectories of artillery shells.[1] For an artillery shot, we would be concerned about the angle of the cannon barrel and the speed of the cannon (if on a ship). We would also want to know the mass of the shell, the amount of powder used, the distance to the target, the wind speed and direction at various altitudes, the friction of the atmosphere, and perhaps even the weather and humidity conditions. Our simulation will be far simpler. We will plot the trajectory of a cannon shell based only on the angle of the cannon barrel and how fast the shell is traveling when it leaves the cannon.[2] We'll assume no friction forces.

If we were to do this problem as a typical exercise in a physics text, the problem would be stated something like this: How far will a cannon shell travel if it has a muzzle velocity of 357 m/s and the angle of elevation of the cannon is 57 degrees above the horizontal? Also, find the highest altitude reached by the cannon shell in this problem?

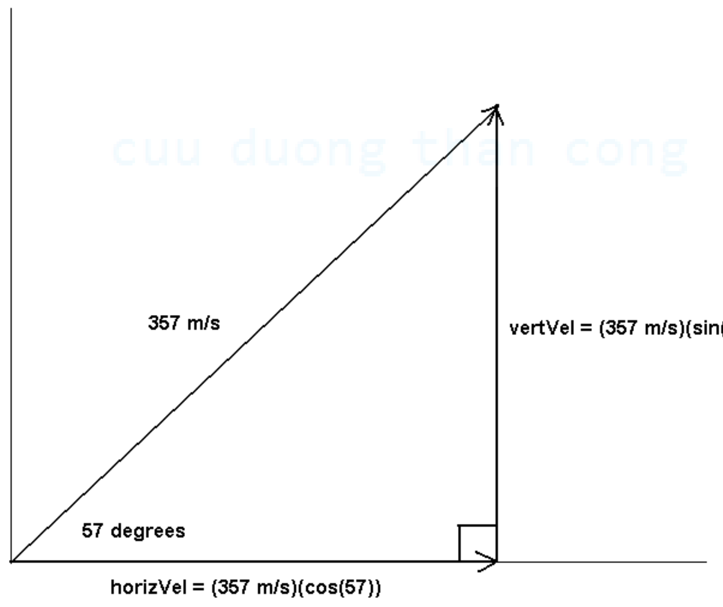


Figure 5.15

Figure 5.15 illustrates the initial conditions in this problem. In order to find a solution mathematically, we would use the following procedure. First, we would realize that we have been given a velocity vector of 357 m/s at 57 degrees above the

[1] Until the electronic computer, trajectories were calculated by hand using a slide rule. The computer automated this process immensely.
[2] The muzzle velocity.

horizontal.[3] Vectors are composed of two parts, a horizontal component and a vertical component. These components are independent of each other; for example, the horizontal velocity of an artillery shell over the earth is not in any way dependent on its vertical movement due to gravity. In other words, gravity does not change regardless of how fast one is moving horizontally and horizontal speed is not affected by gravity.

We have two different problems to solve in this example. The first problem is to calculate how fast the shell is traveling horizontally (with respect to the earth). Once we know this value, how far the shell flies is determined by how long it remains in the air.[4] The calculations are not too difficult, so let's do them "by hand". First, let's determine the horizontal velocity (the **x** component). From Figure 5.15 we see that we'll use the **cos** function. If cos(t) = $\frac{x}{r}$**,** then x = r cos(t).[5] In this case, x, the horizontal velocity, equals 357 m/s (r) times the cos(57).[6] The horizontal velocity of the shell is about 194 m/s. So for every second the shell is in the air, it travels 194 meters horizontally.

The second part of the problem is a bit more difficult. Let's first calculate the initial vertical velocity of the shell (the **y** component). If sin(t) = $\frac{y}{r}$**,** then y = r sin(t). Multiplying 357 m/s times the sin(57)[7] gives us a value of about 299 m/s for the initial vertical velocity of the shell.[8] We are not finished with the vertical **y** velocity, though! We must calculate how long the shell will remain in the air. It starts upward at a rate of 299 m/s, but immediately begins to slow down due to gravity. Eventually, the shell will reach its highest point and for the briefest of times, will have a vertical velocity of zero. It will then begin to fall back to the earth, its velocity increasing each second at a rate consistent with gravity. How do we solve for the time of flight?

We know that the initial vertical velocity is 299 m/s. We know that gravity has an acceleration of -9.8 m/s^2.[9] We also should know that nature exhibits symmetry in our ideal problem… in other words, the first half of the flight is a mirror image of the last half. If our initial vertical velocity is +299 m/s, the final vertical velocity when the shell strikes the ground at the end of its flight will be -299 m/s (notice the sign change).[10] There is a physics formula that states $v_f = v_i + at$… final velocity equals initial velocity plus the acceleration times the time. Solving for time: t = $\frac{v_f - v_i}{a}$. So, t = (-598 m/s)/(-9.8 m/s^2),

[3] Remember, a vector in physics represents both a magnitude and a direction. A scalar represents magnitude only. Velocity is a vector, temperature is a scalar.
[4] Distance = rate x time!
[5] And you thought you would never use algebra?
[6] About 0.545 if we assume the value in the parentheses to be in degrees. The trig functions in Python and other languages normally assume the values representing angles to be in radians
[7] About .839, again assuming the angle is in degrees. More on this later.
[8] The positive values for both x and y indicate motion to the right and upward respectively. Vector direction can be established by +/- signs. Most of the time, a positive number indicates an upward direction for y and motion to the right for x. The opposite sign (-) would indicate motion in the opposite direction. This is defined by humans and not absolute.
[9] The negative sign indicates an acceleration downward toward the center of the earth.
[10] This is why you should NOT fire a gun up into the air. When the bullet comes back down, it is perhaps not moving quite as fast as it was when it left the gun (due to air friction), but it's still moving at a lethal velocity.

or t = 61.0 seconds. The shell is in the air slightly more than a minute. The distance the shell travels is: distance = (horizontal rate) (time) or distance = (194 m/s)(61.0 sec) = 11800 meters (approximately). That's almost 12 kilometers… a bit over 7 miles!

What is the maximum altitude reached by the shell? In physics, there are several ways to solve for this value, but if we use our head, we might make the problem somewhat easier. The total time in the air was 61.0 seconds. This means that it took 30.5 seconds for the shell to climb to its highest point and another 30.5 seconds to fall back to earth. Let's focus on the last half of the journey. The time it took to fall from rest (remember, the vertical velocity at the peak height is zero) back to earth was 30.5 seconds. There is a formula which states that the distance an object travels equals one-half its acceleration times the square of the time, or: $d = 0.5at^2$. In this case, distance = 0.5(-9.8)(30.5*30.5) = -4560 meters.[11] This is a height of over 4.5 kilometers… almost 3 miles!

Well, that was a lot of algebra for a single problem and I included the mathematics in the text for comparison purposes. Since we are interested in programming, the question becomes "Can we write a Python program that will not only do the proper calculations for us, but also display the flight of the shell (the trajectory)?" Of course we can! What should our program look like when it runs? The program should ask for two inputs in the terminal console. The first input should be the angle of elevation of the cannon barrel in degrees. The second input should be the muzzle velocity of the shell in meters per second (m/s). We then want the code to solve the problem and plot the trajectory of the projectile[12] in an OpenGL window. By the way, if you were thinking that the solution of this problem looked a bit like the parametric equations we studied in the last section, you are thinking correctly! We used two sets of parametric equations here in this situation. The first set of parametric equations used the cannon barrel angle measurement to determine the individual velocities of **x** and **y**. The second set of equations used **t** (time) to solve for flight duration and distance traveled. We are actually using calculus in this program to solve our trajectory problem. Now that's impressive... be sure to tell your math teacher!

When we solved the physics problem algebraically, we found a solution for a single set of angle and velocity parameters. Unless we are willing to painstakingly create a table of elevation angles and muzzle velocities,[13] we will have to algebraically solve each new problem in the same manner. Our computer program must be flexible and able to solve any reasonable problem of the same type without major code revisions. This program is the most ambitious we’ve tackled thus far. We will build on the previous programs and add many new features. Be very careful to type in the code without errors. Save the program as **pycannon.py** or something similar.

Here is the program listing for **pycannon.py**. As usual, pay CLOSE attention to the indentations!

```
# PyCannon.py
# A Physics Simulation
```

[11] The negative sign means that we are measuring downward. Distance is a vector, having both magnitude and direction. It's still 4560 meters high!
[12] A fancy "more official" name for an object obeying gravity and the laws of physics.
[13] Such tables were created for wartime use by both human computers and electronic computers.

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

global horizvel
global vertvel

def init():
     global horizvel
     global vertvel

     # White background
     glClearColor(1.0, 1.0, 1.0, 1.0)

     # Large range for long shots
     gluOrtho2D(-200.0, 12000.0, -200.0, 5000.0)

     # Input angle and muzzle velocity
     angle = input("Enter the angle of elevation: ")
     muzzvel = input("Enter the muzzle velocity of the shell: ")

     # Convert the degree angle to radians
     radangle = (angle*3.1415926)/180

     # Solve for horizontal and vertical initial velocities
     horizvel = muzzvel*cos(radangle)
     vertvel = muzzvel*sin(radangle)

     # Print out the initial velocities in the console
     print
     print ("Horizontal Velocity (m/s) = "), horizvel
     print ("Vertical Velocity (m/s) = "), vertvel

def plottrajectory():
     global vertvel
     global horizvel

     # We can now calculate and change vvel
     # While preserving the original vertvel
     vvel = vertvel
     hvel = horizvel

     glClear(GL_COLOR_BUFFER_BIT)
     glColor3f(0.0, 0.0, 0.0)

     # Draw some horizontal and vertical axis lines
     glLineWidth(2.0)
     glBegin(GL_LINES)
     glVertex2f(0.0, 0.0)
     glVertex2f(20000.0, 0.0)
     glVertex2f(0.0, 0.0)
     glVertex2f(0.0, 15000.0)
     glEnd()

     # Set the height of the cannon barrel
     # Initalize variables for later use
     height = 2.0
     dtime = 0.0001
     dist = 0.0
     maxheight = 0.0

     # Plot the trajectory as long
     # as the height is above the ground
     while height > 0.0:

               # Equations to calculate distance and
               # Height for each unit of time.
               dist = dist + hvel*dtime
               vvel = vvel - 9.8*dtime
               height = height + vvel*dtime

               # Find the max height.
               if maxheight < height:
                    maxheight = height

               # Plot the trajectory
               glBegin(GL_POINTS)
               glVertex2f(dist, height)
               glEnd()
     glFlush()

     # Print the solutions.  Not indented!
     print
     print("Distance traveled (m) = "), dist
     print("Maximum altitude (m) = "), maxheight

def main():
     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE)
     glutInitWindowPosition(50, 50)
     glutInitWindowSize(800, 600)
     glutInit(sys.argv)
     glutCreateWindow("How Far Will It Go?")
     glutDisplayFunc(plottrajectory)
     init()
     glutMainLoop()

main()
# End Program
```

Save and run the program. Use the values of 57 for cannon angle and 357 for the muzzle velocity to replicate the problem we solved by hand earlier in this section.

You should see something similar to Figure 5.16 if all went well. The graph looks like a parabola, which shouldn't be too surprising if you have ever watched the flight of a thrown or batted ball. Perhaps we've "discovered" a parametric representation of a parabola?

This program is longer and more complex than our previous efforts. You should notice that I've made an attempt at using more verbose comments than in previous listings. The **`import`** lines are the same as in earlier programs. However, we see two new lines:

```
global horizvel
global vertvel
```

Variables in Python are usually local, meaning that they are defined only in the function in which they are used. For example, in the **`def init():`** function, the variables **`angle`**, **`muzzvel`**, and **`radangle`** are used for calculations, but they are valid only in this function. The **`plottrajectory`** function knows nothing of these three variables! If we need to use a variable throughout the program, we need to declare the variable as a **`global`** variable.[14] Here, we are using two **`global`** variables, **`horizvel`** and **`vertvel`**. These variables will store the horizontal and vertical velocities respectively. I anticipate needing to use the values stored in these variables in other places, hence the **`global`** declaration.

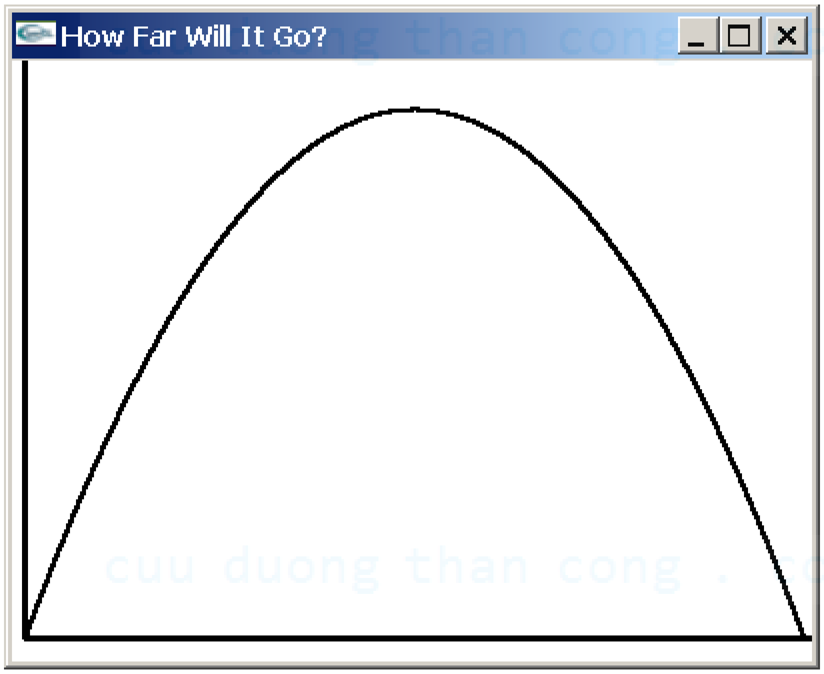


Figure 5.16

[14] Python experts frown on using **`global`** variables, but… well, the experts aren't here now, are they? I think **`global`** variables are handy in simple situations such as this program. The problem is that Python makes it a bit difficult to declare them properly and if too many **`global`** variables are used in a large program, they can be easy to forget, misplace, or misuse.

The program closely follows the algebraic solution given at the beginning of this section and remark statements are used to help clarify many of the code statements. In the **`def init():`** function, we first declare (again) the global variables **`horizvel`** and **`vertvel`**. We must do this in each function that assigns or changes the values of these variables. The **`glClearColor`** command is not new, but notice in **`gluOrtho2D`** how the range has been changed considerably to reflect the nature of this problem. We then use an **`input`** statement to store the cannon elevation angle (in degrees) and muzzle velocity in the variables **`angle`** and **`muzzvel`**. The trig functions needed to calculate the horizontal and vertical components of the muzzle velocity can't use angle measures in degrees, so we must convert degrees to radians using the standard conversion:[15]

```
radangle = (angle*3.1415926)/180
```

Next we solve for both the horizontal and vertical components of the muzzle velocity vector. These components serve and the initial horizontal and vertical velocities in the problem. The horizontal velocity remains constant since there is no friction in our ideal world. The vertical velocity changes due to the influence of gravity.

```
horizvel = muzzvel*cos(radangle)
vertvel = muzzvel*sin(radangle)
```

As an added touch, we print the values of **`horizvel`** and **`vertvel`** in the console window at the end of the **`init()`** function. The "empty" **`print`** statement simply inserts a blank line for "pretty print".

In the **`def plottrajectory():`** function, we again declare the global variables **`horizvel`** and **`vertvel`**. We also add the lines:

```
vvel = vertvel
hvel = horizvel
```

The vertical velocity of the cannon shell will be changing due to gravity. If we resize the graphics window, the trajectory will need to be redrawn. If we don't preserve the original value of **`vertvel`**, our solution and plot will be incorrect. By assigning **`vvel`** the value of **`vertvel`**, we can keep the initial vertical velocity stored in **`vertvel`** "safe" for future use. Similarly, we need to preserve the initial horizontal velocity in **`horizvel`** for the same reason; we may need to use it again!

The next few lines are business as usual, except we are moving the origin to the lower left corner and extending the axis lines to reflect the extended range. We then declare some local variables:

```
height = 2.0
dtime = 0.0001
dist = 0.0
maxheight = 0.0
```

---

[15] Radians = $\frac{\deg * \pi}{180}$

The variable **`height`** is the initial height of the cannon barrel above the ground (in meters). **`dtime`** is the time slice used to calculate the trajectory of the shell.[16] We then set the **`dist`** (distance) and **`maxheight`** variables to zero. This serves to declare the variables so they can be used properly and also assigns them an initial value.

We now let Python calculate the solution:

```
while height > 0.0:

     # Equations to calculate distance and
     # Height for each unit of time.
     dist = dist + hvel*dtime
     vvel = vvel - 9.8*dtime
     height = height + vvel*dtime

     # Find the max height.
     if maxheight < height:
          maxheight = height

     # Plot the trajectory
     glBegin(GL_POINTS)
     glVertex2f(dist, height)
     glEnd()

# End of while loop
```

As long as the height of the shell is above the ground (**`while height > 0.0:`**), we continue calculating the distance (**`dist`**), vertical velocity (**`vvel`**), and **`height`**. The equations may look a bit strange, but let's think about them a bit. In the **`dist`** equation, we simply take the current distance (where the shell is at that instant) stored in the variable **`dist`** (on the right side of the "**`=`**") and add to this value the additional distance calculated by **`hvel*dtime`** (rate x time). This cumulative value, the current distance plus the new distance, is then reassigned to the **`dist`** value on the left side of the "**`=`**". This is how we keep a running total of the horizontal distance traveled. The vertical velocity (**`vvel`**) is calculated in a similar fashion. The current vertical velocity (**`vvel`**) on the right side of the "**`=`**" is modified by **`-9.8*dtime`** (-acceleration x time) and the new value is assigned to **`vvel`** on the left side of the "**`=`**". The height above the ground is calculated by using the current **`height`** (on the right side of the "**`=`**") and modifying this value by adding **`vvel*dtime`** (rate x time). The new **`height`** is then assigned to the "**`height`**" variable on the left side of the "=". If the term "iteration" came to you during this discussion, then you are a true computer geek[17]… this process definitely illustrates the concept of iteration. We are using the output from one set of calculations as input for the next set (notice that the variables **`vvel`** and **`height`** appear

[16] "**`dtime`**" is a crucial variable. We are solving this problem by dividing it into many small pieces and then adding up the results of each piece to provide a total solution. The size of "**`dtime`**" determines the size of the "piece" of the problem. This numerical method can be quite accurate depending on the size of the time slice or "piece". We are actually doing calculus here!
[17] I mean this in a nice way, of course! It is absolutely a compliment to be called a "geek".

on BOTH sides of the "**=**") and repeating the process until the shell height is 0.0, meaning that the shell has hit the ground!

How do we find the maximum height (**maxheight**)? As the shell is going upward, each new position is the maximum height, so we can set the **maxheight** variable equal to **height** (**maxheight = height**). We do this by checking to see if the new **height** is greater than the old **maxheight** (**if height > maxheight:**). If this is true, and it always will be as the shell is rising, then we set **maxheight = height**. As we pass the very top of the trajectory, the **height** begins to decrease and will never again be greater than the "old" **maxheight** variable. The **maxheight** variable will no longer change because the **maxheight = height** statement will be ignored (**if height > maxheight:** is now false... **height** is LESS than **maxheight** because the shell is on the way down). As a result, we have preserved the value of the maximum height in the **maxheight** variable. At the end of the code block the points corresponding to each **(dist**, **height)** ordered pair are plotted using the **glVertex2f** statement, creating the parabolic trajectory.

The **print** statements that display the numerical results of the problem calculations are not indented at the same level as the other lines in the **plottrajectory** function. This insures that these lines are NOT in the **while** loop and only display the final results at end of the function when the calculations are complete.

In order to obtain a computer solution to a problem that unfolds over time, such as the projectile example here, we must simulate as closely as we can the conditions of the problem. For all practical purposes, time is a continuous process. This means that there are no "breaks" in the flow of time and that no matter how finely we divide a unit of time, we will never reach a point where time doesn't exist… in other words there are no holes in a real timeline! Computers are unable to handle such a number system. There is a practical limit to the precision of computer arithmetic. Eventually, if we keep dividing a number, we'll reach a point where our computer can no longer distinguish between two adjacent values. This limit is usually around 15 or 16 decimal places.[18] In English, this means that there are "holes" in the computer number line.[19] How do we reconcile the real world arithmetic (which is infinitely precise) with the computer world arithmetic (which is "holey")? We do the best we can. In this problem, we divide up the time flow of the problem into the smallest units of time we can, given the speed of our computer. In this way, we can approximate behavior of a real projectile under ideal conditions.[20]

[18] Depending on the computer, the software, the programmer, etc. The more decimal places, the longer it takes to make a calculation. If we want our computer to do "real-time" simulations, we can't take much time for an individual calculation. Hence the trade-off between speed and precision.
[19] Assume that your computer can only represent 3 decimal places. Your computer could not tell you the difference between 0.3210123 and 0.3210321. This may not seem important, but as we'll see later, this difference is huge when it comes to real-world simulations.
[20] The time-slice method works this way: the first time step or slice uses the initial conditions for the whole problem… zero time and the initial values for all variables. The program then solves the problem for this small time slice. The solution to the first time slice becomes the initial conditions for the second time slice and the computer then solves that problem. The solution for the second time slice becomes the initial conditions for the third… and so on. The final answer is

You can think of it this way. We are slicing (using the **`dtime`** “knife”) the problem up into hundreds, thousands, or even millions of sequential pieces. The computer solves each individual “piece” and sums the total for the approximate answer to the problem. The tinier the slice (the smaller the **`dtime`** variable) the more closely we mimic nature and the more accurate our answer. The problem is that if we make **`dtime`** too small, the computer simulation takes longer to run… much longer, possibly, than actually doing the real experiment. We must find a balance between runtime and accuracy.[21] You might think that using computers to solve equations isn’t such a good thing when we can get more precise (possibly) answers by solving the equations manually or with a computer algebra system such as Mathematica. However, many (most?) equations involved in physical models can’t be solved easily or perhaps not at all using manual (analytic) methods. In that case, we have no choice but to use a computer.

The moral of the story is that computers can be very flexible problem-solving devices. When we solve a physics example by hand, we employ analytical methods and find an algebraic solution. Computers, if programmed properly, can do this as well. In this example, we used both analytical methods (to solve for initial horizontal and vertical velocities) and numerical methods (to calculate running totals for distance, vertical velocity, height, and maxheight). Numerical methods follow a recipe or algorithm to arrive at solutions to problems. This may not seem important, but many equations simply can't be solved by any other method. In any case, computers are very valuable tools for calculating solutions to problems.

When working these exercises, you may want to save the original program from this chapter under a different name and work with the newly named version. That way you preserve the original in the event you need it later. Most of the modifications required in the exercises are easily reversed by simply assigning variables to their original state or removing the new lines of code.

Exercises

1) Try several different problem scenarios. You may have to adjust the ranges in **`gluOrtho2D`** to display the results.

2) Change the gravitational constant in **`vvel = vvel - 9.8*dtime"`** in the **`plottrajectory`** function to **`-1.6`** (keep the "**`-`**" sign in place). This will simulate gravity on the moon. You will probably have to adjust the plot ranges in **`gluOrtho2D`**![22]

3) In exercise 2, change the sign from "-" to "+" and see what happens. Before you run the program, see if you can predict the outcome. Were you correct?

4) This exercise will be a challenge. For every muzzle velocity, there is a maximum horizontal distance a projectile will travel. Try an angle of 45 degrees and a muzzle

---

the summation of the solutions for all the small time slices. Each plotted point of the trajectory is a solution to a particular time slice. Again, this is calculus!

[21] Or you use a super-computer, computer cluster, or grid.

[22] Also adjust the axis line ranges in the **`glBegin(GL_LINES)`** section of code!.

velocity of 350 m/s. Write down the distance the projectile travels. Now try an angle of 40 degrees with the same muzzle velocity. Try an angle of 60 degrees with the same muzzle velocity. What do you notice? Now here is the challenge. Can you find pairs of different angles that produce the same distance with the same muzzle velocity? Is there a pattern to these "double angles" that would allow you to predict a method that, when given one angle, you can find the other?

5) Try various values for **`dtime`**. What happens when you reduce the value of **`dtime`** for a shorter time interval? What happens when you increase the value of **`dtime`** for a longer time interval? Which is best for solving the problem, a very short time interval (**`dtime`** < 0.0001) or a longer time interval? Why?

6) You may want to try different plot colors, line widths, and point sizes for the program. Also experiment with the background color. What combination of settings provides the most pleasing display?

7) Calculate the distance an object (such as a bowling ball) would travel horizontally if it drove off a vertical cliff at a certain velocity. Say we have an a bowling ball traveling at 50 m/s and it flies off a 1000 m cliff. How far away from the cliff would the ball strike the valley floor? Hint: Look at the **`height`** variable to take the altitude of the cliff into consideration and consider what the angle of elevation should be.

8) An addition to problem 7 would be to print vertical velocity of the auto as it strikes the ground. Where would you place this **`print`** statement in the program and what variable would you use?

9) Research beowulf clusters, grid computing, and supercomputers. What are the possible uses of such computer systems? Why might they be valuable to us based on what you have learned in this section? Google "MPICH Blank" if you are interested in building your own private beowulf cluster.[23]

10) Sometimes a physics problem will ask for the total amount of time an event takes to complete. It would be nice to have our program calculate the total time the shell (or bowling ball) is in the air. You might think that the variable **`dtime`** contains this information, but this is not correct. "**`dtime`**" contains the length of the time step or slice… remember, we are breaking this problem up into many small individual problems and summing them for a final solution. Here's more than a hint: if we can add up the number of time slices in the problem, we can calculate a total time of flight. Place the line: **`totaltime = 0.0`** immediately underneath the section where we assign initial values to **`height`**, **`dtime`**, **`dist`**, and **`maxheight`**. The variable **`totaltime`** will store the total time of flight, so we need to define **`totaltime`** and initialize it to **`0.0`**. Place the statement **`totaltime += dtime`** in the **`while height > 0.0:`** loop immediately after the line **`height = height + vvel*dtime`**. Finally, we need to print the value of **`totaltime`**. I'll leave that up to you… but make certain you follow the format of the other **`print`** statements by providing information about the value you are displaying.

[23] Yeah, that's me.

11) Redo exercise 7 using the **`totaltime`** concept in exercise 10. This will tell you how long it takes to fall a certain height (under ideal conditions). Using symmetry, you can determine this information from a ground launched projectile (as in the original problem) by dividing the total flight time in half.

12) The problem as stated involves an ideal situation… we are assuming no friction. How could we simulate the effects of friction? At first thought, we might try subtracting a small value from the horizontal and vertical velocities. Let's see how this works. Change the **`while height > 0.0:`** loop as follows, noting the addition of **`hvel -= 0.001`** and **`vvel -= 0.001`**. Both of these modifications subtract **`0.001`** from the motion (we think?) of the cannon shell, hopefully(?) behaving like friction. Use the original values of 57 degrees elevation and 357 m/s muzzle velocity.

```
while height > 0.0:

     # Equations to calculate distance and
     # Height for each unit of time.
     dist = dist + hvel*dtime
     vvel = vvel - 9.8*dtime
     height = height + vvel*dtime
     hvel -= 0.001
     vvel -= 0.001

     # Find the max height.
     if maxheight < height:
          maxheight = height

     # Plot the trajectory
     glBegin(GL_POINTS)
     glVertex2f(dist, height)
     glEnd()

# End of while loop
```

Here's the plot in figure 5.17

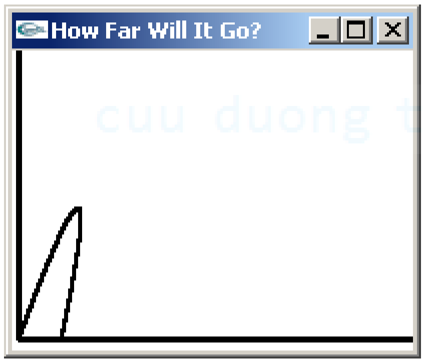


Figure 5.17

That's some strong wind blowing from right to left... only the wind isn't blowing! Friction opposes motion, it doesn't cause motion.[24] Our model is incorrect. It turns out that friction in a fluid is much more complicated than our simple **`0.001`** subtraction. Friction only works when an object is in motion. Once an object reaches zero velocity, friction will not make it move less! So, friction depends on motion and we need to reflect this idea in our model. Modify the **`hvel`** and **`vvel`** "friction" lines as follows:

```
hvel -= .00000002*hvel**2
if vvel > 0.0:
     vvel -= 0.00000002*vvel**2
else:
     vvel += 0.00000002*vvel**2
```

In this new model, the friction depends on the motion in **`hvel`** and **`vvel`**.[25] If either becomes zero, then friction also becomes zero. You may wonder why we've added an **`if..else`** statement? First, in the **`hvel`** equation, friction is always negative because the projectile always moves toward the right with a positive velocity (**`hvel > 0.0`**). With the **`vvel`** equations, **`if vvel > 0.0:`** (the shell is going upward), then we need to subtract from the positive **`vvel`** value. If **`vvel`** is negative (**`vvel`** < 0.0), then the **`else`** statement is called and we ADD a positive expression to **`vvel`**. I know this sounds confusing, but we must make certain that friction always opposes (has the opposite sign of) motion. There are times when the **`vvel`** is positive (going up!) and there are times when **`vvel`** is negative (falling down!). The two **`vvel`** statements in the conditional block serve the purpose of always opposing the motion of **`vvel`**.[26] The value of 0.00000002 is "made up" in that I simply wanted to provide a constant that represented[27] the density/viscosity of air.

Run the model again using the same initial values and compare the results of figure 5.18 on the next page with figures 5.17 and 5.16. You can see that the trajectory is definitely not a perfect parabola as in figure 5.16 and the projectile is not "blown" backwards as in figure 5.17. You'll also note that the maximum height and distance are considerably reduced. The new model still doesn't perfectly portray friction, but it's better than the first effort. In science, models are useful to the extent that they accurately portray nature. A model aircraft in a wind tunnel is not the same thing as the real airplane in flight, however we still may learn something useful from the model. We might be able to say the same thing about our projectile/air friction model here. The simulation (virtual cannon shell) is not the same as a real cannon or howitzer shell, but our model may serve to help us learn something about the behavior of projectiles in a fluid.

[24] If you were to create a tank or artillery game, you might need the idea of "wind" for game play. The wind would act somewhat like the constant value we are subtracting from **`hvel`**.
[25] The velocities stored in **`hvel`** and **`vvel`** are squared to be somewhat realistic. It actually is true that if you double your velocity, then air friction is 4 times as great. Tripling your velocity multiplies air friction by 9, etc.
[26] Which is what friction does!
[27] Conceptually, if not accurately.

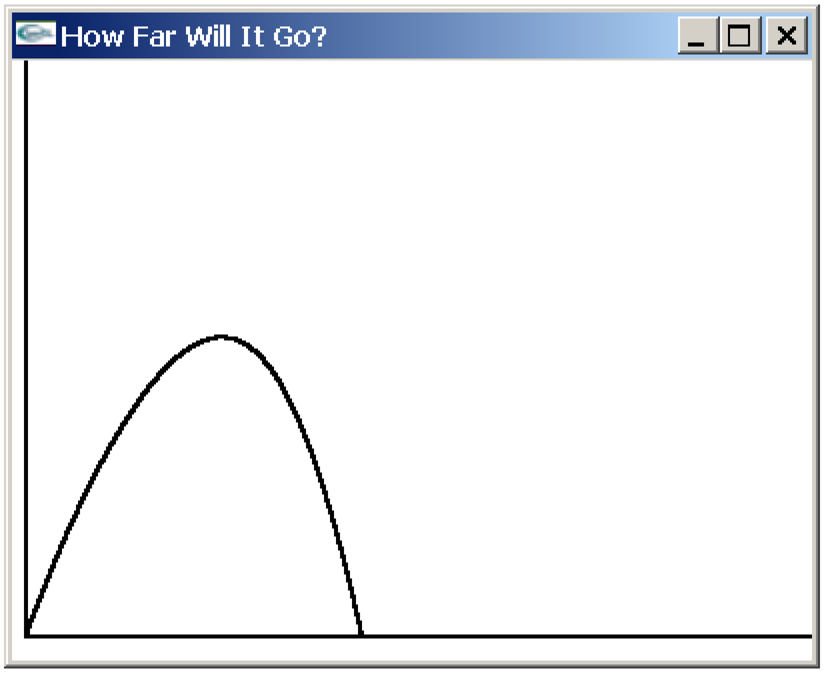


Figure 5.18

Experiment with different cannon elevation angles and muzzle velocities. For example, try an elevation of 25 degrees and a muzzle velocity of 975 m/s. How does your plot compare to figure 5.19 on the next page? It looks a lot like the profile of a seven iron shot.[28] This should tell us that even though the model is not perfect, we are at least making a decent attempt at a simulation of air or fluid friction.

Experiment with your own friction equations or with those you have researched. Change the value of the air density/viscosity constant of 0**.00000002**. What happens if you increase this value? What happens if you decrease this value? No matter how you change the value of this constant, the new equations based on the square of the velocities guarantee that the projectile will not fly the "wrong" way as in figure 5.17. Once either **`vvel`** or **`hvel`** become zero, the friction associated with that particular velocity also becomes zero (as it should!).

A major part of learning to program depends on your own efforts at modifying programs and experimenting with new ideas.

As a challenge, research fluid friction and see if you can improve the model![29]

[28] For the golfers in the crowd… but seven irons don't travel nearly this far! Att least mine don't.
[29] This is a tough problem! Best wishes…

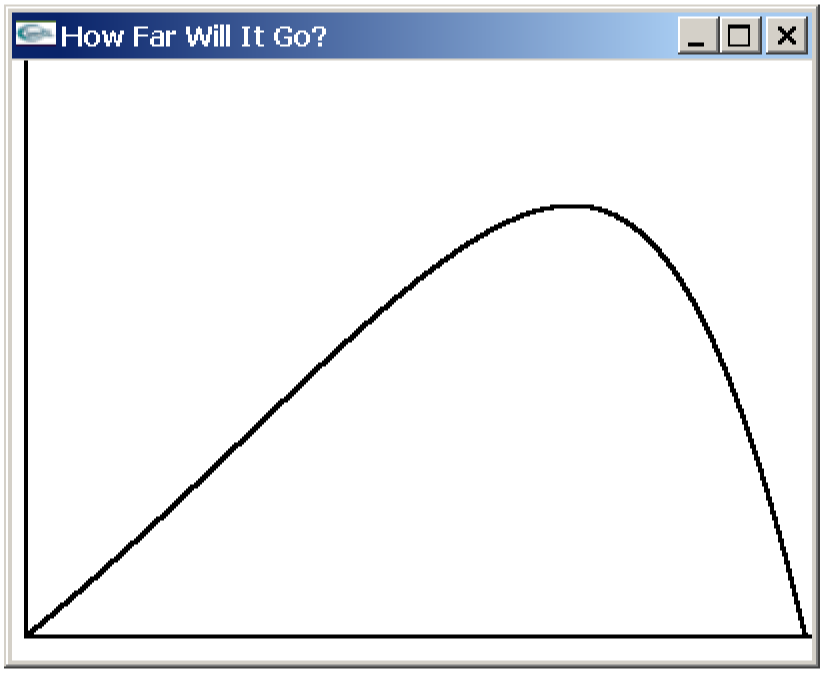


Figure 5.19

13) Research the concept of difference equations. This program uses difference equations to solve for the trajectory of the projectile. Difference equations are powerful computational tools for providing numerical solutions to difficult equations. Difference equations also are a topic in calculus. You are doing some rather sophisticated mathematics in this chapter!

Section 5.5 Polar Coordinates

In algebra, we learned to plot points and graph equations using the Cartesian rectangular coordinate system. This coordinate system, the brainchild of Rene Descartes, established a link between points in a plane and ordered pairs of real numbers. In sections 5.1 and 5.2, we plotted individual points and graphed 2D functions using the traditional x-y Cartesian coordinate system. In section 5.3, we used parametric equations to plot Lissajous/Bowditch curves… again using the Cartesian system. Other coordinate systems also exist and we are not limited to Cartesian coordinates.[1]

One such system is the use of polar coordinates to map points on the plane. We begin by establishing a fixed point, which we can refer to as the origin[2] or pole. If we then determine a length from the origin and an angle from the zero axis, we can plot any point in the plane. The ordered pair looks like this: (r, θ), where "r" is the length or radius of the point and θ is the angle from the zero π radian axis. Figure 5.17 illustrates a point ***P*** in the polar coordinate system.

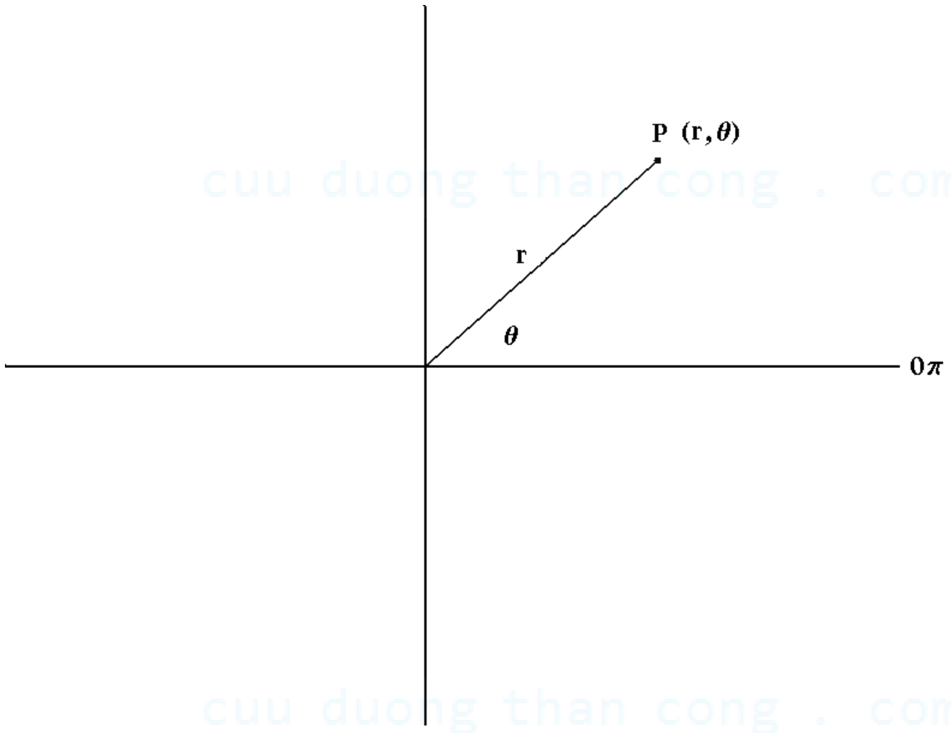


Figure 5.20

[1] Well, actually we are limited in computer graphics. The screen coordinates map nicely to (x,y) ordered pairs. We CAN calculate using other systems and convert to screen coordinates. That is what we'll do in this section.
[2] The origin is a reference point in the Cartesian system. We need a reference point in any coordinate system we use.

Graphs in the polar coordinate system look something like: **r = 4cosθ + 2**. How do we plot them on a computer screen? We have to convert this equation into Cartesian coordinates![3] We already know how to do this, though, from our parametric equation work. If you'll recall, in the parametric format we assumed a radius of **r = 1**. In polar equations, we'll not make this assumption and allow **r** to vary according to some rule or equation. The conversions from polar to Cartesian coordinates are:

x = r·cos(θ)
y = r·sin(θ)

When we combine these two conversions with an equation for r, such as **r = 4cosθ + 2**, interesting things happen! As an analogy, you can think of a polar equation as a fancy "radar" screen like you may have seen in movies featuring military radar tracking incoming planes or missiles.[4] The **r** value sweeps around according to angle **θ.** The variable **r** can shrink or lengthen according to a rule such as **r = 4cosθ + 2**. Together, **r** and **θ** produce a polar graph. Enough talk; let's try some Ruby code to plot polar equations. New in this program will be a "keyboard" function that will allow us to interact with the program and a "reshape" function that will help maintain the proper shape[5] of the graphics.

Type in the listing below and save the program as **pypolar.py** or something similar. Again, pay close attention to indentations and spelling!

```
# PyPolar.py
# Plotting Polar Equations

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

#  Set the width and height of the window with global variables
#  Set the axis range globally using global variable axrng
global width
global height
global axrng

#  Initial values
width = 400
height = 400
axrng = 7.0

def init():
     glClearColor(1.0, 1.0, 1.0, 1.0)
```

[3] See footnote 1 in this section again.
[4] This was also used as an analogy in the parametric equation section.
[5] Also called the aspect ratio of the graphic figure. You may have noticed that if you resized or maximized the graphics window in earlier programs, the plot figure will resize, but will look distorted or stretched. We'll fix this problem with a new function called **def reshape:**.

```
# GLUT Display Function
def plotpolar():
     glClear(GL_COLOR_BUFFER_BIT)

     # Plot axis lines for reference
     glColor3f(0.0, 0.0, 0.0)
     glBegin(GL_LINES)
     glVertex2f(-axrng,0)
     glVertex2f(axrng,0)
     glVertex2f(0,axrng)
     glVertex2f(0,-axrng)
     glEnd()

     # Plot polar equation for a Limacon
     glPointSize(2.0)
     glBegin(GL_POINTS)
     for theta in arange(0.0, 6.28, 0.001):
          r = 4*cos(theta) + 2
          x = r*cos(theta)
          y = r*sin(theta)
          glVertex2f(x,y)
     glEnd()
     glFlush()

# This is new... this is a reshape function so that the
# aspect ratio of the graphics window will be preserved
# and anything we draw will look in proper proportion
def reshape(w, h):

     # To insure we don't have a zero window height
     if h==0:
          h = 1

     #  Fill the entire graphics window!
     glViewport(0, 0, w, h)

     #  Set the projection matrix... our "view"
     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     #  Set the aspect ratio of the plot so that it
     #  Always looks "OK" and never distorted.
     if w <= h:
          gluOrtho2D(-axrng, axrng, -axrng*h/w, axrng*h/w)
     else:
          gluOrtho2D(-axrng*w/h, axrng*w/h, -axrng, axrng)

     #  Set the matrix for the object we are drawing
     glMatrixMode(GL_MODELVIEW)
     glLoadIdentity()
```

```
def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'
     if key == chr(27):
          sys.exit()
     if key == "q":
          sys.exit()

def main():
     global width
     global height

     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
     glutInitWindowPosition(10,10)
     glutInitWindowSize(width,height)
     glutCreateWindow("Polar Equations")
     glutReshapeFunc(reshape)
     glutDisplayFunc(plotpolar)
     glutKeyboardFunc(keyboard)

     init()

     glutMainLoop()

main()

# End Program
```

If everything is typed correctly and the program runs without error, you should see something like Figure 5.21:

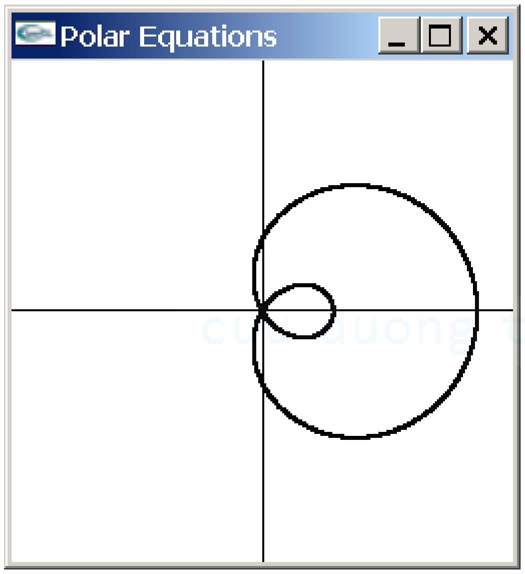


Figure 5.21

Figure 5.21 is a nice rendering of a looping curve called a Limaçon. In the exercises at the end of this section we'll explore several "famous" polar equations.[6]

Now we'll look at the code for Figure 5.21 a bit more closely.[7] The first few lines are the familiar **`import`** statements required of all OpenGL programs, including the **`Numeric`** module for enhanced mathematical functions. They are followed by:

```
#  Set the width and height of the window with global variables
#  Set the axis range globally using global variable axrng
global width
global height
global axrng

#  Initial values
width = 400
height = 400
axrng = 7.0
```

We are using global variables to set the initial **`width`** and **`height`** of the graphics window. Also, we are using a variable called **`axrng`** to establish the **`x`** and **`y`** axis ranges. **`axrng`** also will also be used by the **`gluOrtho2D`** statement in the **`def Reshape( w, h):`** function. Following the **`global`** variable declarations, the initial values of these variables are set. We are choosing a graphics window 400 x 400 pixels and an **`axrng`** (axis range) of 7.0. Any changes made to these variables will result in a "global" change of their values… far easier than searching through the code and manually typing in new numbers!

Now for the **`def plotpolar():`** display function:

```
def plotpolar():
     # Clear the screen
     glClear(GL_COLOR_BUFFER_BIT)

     # Plot axis lines for reference
     glColor3f(0.0, 0.0, 0.0)
     glBegin(GL_LINES)
     glVertex2f(-axrng,0)
     glVertex2f(axrng,0)
     glVertex2f(0,axrng)
     glVertex2f(0,-axrng)
     glEnd()

     # Plot the polar equation for a Limacon
     glPointSize(2.0)
     glBegin(GL_POINTS)
     for theta in arange(0.0, 6.28, 0.001):
```

[6] And perhaps some not so famous curves as well.
[7] As the text progresses, I'll highlight only the new concepts or the use of "old" concepts in a new way. I will not discuss every single line in detail because you are past the complete novice stage by now! Also, I will begin using more remark statements to help explain the code.

```
            r = 4*cos(theta) + 2
            x = r*cos(theta)
            y = r*sin(theta)
            glVertex2f(x,y)
        glEnd()
        glFlush()
```

The **def plotpolar():** function is not radically different from previous programs, but notice the use of the **axrng** variable in drawing the axis lines within the **glBegin(GL_POINTS)** section. It simply makes sense to draw axis lines based on the ranges of the x and y axes. The other changes are found within the same **glBegin(GL_POINTS)** code segment. We are defining the variables **r**, **x**, and **y** by equations using **theta** (θ). It is here, of course, that the polar graphics "picture" is created and plotted.

As stated in an earlier footnote, when the size of the screen is manually changed by using the mouse, either by maximizing the windows or by dragging the edges or corners, the drawing may become distorted. We will now "fix" this problem by using a new function: **def reshape(w, h):**[8]

```
def reshape(w, h):

        # To insure we don't have a zero window height
        if h==0:
            h = 1

        #  Fill the entire graphics window!
        glViewport(0, 0, w, h)

        #  Set the projection matrix... our "view"
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()

        #  Set the aspect ratio of the plot so that it
        #  Always looks "OK" and never distorted.
        if w <= h:
            gluOrtho2D(-axrng, axrng, -axrng*h/w, axrng*h/w)
        else:
            gluOrtho2D(-axrng*w/h, axrng*w/h, -axrng, axrng)

        #  Set the matrix for the object we are drawing
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()

# End Reshape
```

You've no doubt noticed the "**w**" and "**h**" in between parentheses in the **reshape** function name? A quick glance at **def main():** will show that we've actually defined the **reshape** function using the **glutReshapeFunc(reshape)** command. GLUT and

[8] Again, we try to give functions names that make sense.

OpenGL both now know that any time the graphics window is resized, the **`reshape`** function will be notified or called. The variables **`w`** and **`h`** are "passed" to the **`reshape`** function by OpenGL/GLUT and are "caught" by the **`(w, h)`** variables in the function name. **`w`** and **`h`** now contain the new values for the width (**`w`**) and height (**`h`**) of the graphics window. This is great! We can use this new information to reset the drawing "canvas" using **`gluOrtho2D`** so that our plot image is not distorted!

The lines:

```
if h==0:
     h = 1
```

keep us from making the resized window of zero height. This is important because we'll be using **`h`** as a divisor later. The **`glViewport(0, 0, w, h)`** command makes certain that the entire graphics window contains the display. It is possible to use this command to "peek" at small sections of a graphic plot within a window, but we want to view the entire picture. Note the use of **`w`** and **`h`** again!

As our programs become more complex, we become concerned about two viewing concepts. The first is how we are going to draw the object itself and the second is how we are going to place and view the object within the "world" contained by the graphics screen. The next two lines concern themselves with the viewing "world" where the object will eventually be placed.

```
#  Set the projection matrix... our "view"
glMatrixMode(GL_PROJECTION)
glLoadIdentity()
```

A **`glMatrixMode()`** command is usually followed by **`glLoadIdentity()`**. The **`GL_PROJECTION`** parameter specifies the world view or projection matrix. If you haven't had linear algebra, you can think of a matrix as a system of equations that govern the **`x`**, **`y`**, and **`z`** coordinates of our graphics world. Doing mathematics with these equations (the projection matrix)[9] manipulates how we see the world. We can translate (or move laterally) the world, rotate the world, and make the world larger or smaller (scale the world).

```
#  Set the aspect ratio of the plot so that it
#  Always looks "OK" and never distorted.
if w <= h:
     gluOrtho2D(-axrng, axrng, -axrng*h/w, axrng*h/w)
else:
     gluOrtho2D(-axrng*w/h, axrng*w/h, -axrng, axrng)
```

Once we have the projection matrix set to the identity matrix (using **`glLoadIdentity`**)[10] we can then set the new **`x`** and **`y`** ranges using **`gluOrtho2D`**. We

[9] Yes, I've seen the movies. No, it isn't quite the same thing.
[10] **`glLoadIdentity()`** resets the projection or modelview matrices back to the identity matrix… which serves as a starting point for all future calculations. This is important for proper graphics rendering.

check to see if the width is less than or equal to the height and choose the correct **gluOrtho2D** command based on this information (using the **if** statement). What do you think **else:** means here? **axrng** is used once again to establish the new **x** and **y** axis ranges and, more importantly, "fix" the distortion caused by stretching, shrinking, or maximizing the window. Where do you think the "fix" occurs and how are **w** and **h** used? What we are doing here is changing the aspect ratio of the graphics window according to the new window dimensions.

Once we have set the new window ranges and aspect ratios, we are ready to redraw our plot. Since we want to draw our object or "model" again, we set the **glMatrixMode()** to the **GL_MODELVIEW** matrix. Note the **glLoadIdentity()** after **glMatrixMode(GL_MODELVIEW)**. **glLoadidentity()** resets the **GL_MODELVIEW** matrix so that all future translations, rotations, and scalings begin with the identity matrix.

```
#  Set the matrix for the object we are drawing
glMatrixMode(GL_MODELVIEW)
glLoadIdentity()
```

The **reshape** function is very useful because it can be inserted into nearly any program. To see how this function operates, let's look at two examples. Figure 5.22 demonstrates a resized graphics window (from figure 5.21) WITHOUT using the **reshape** function.

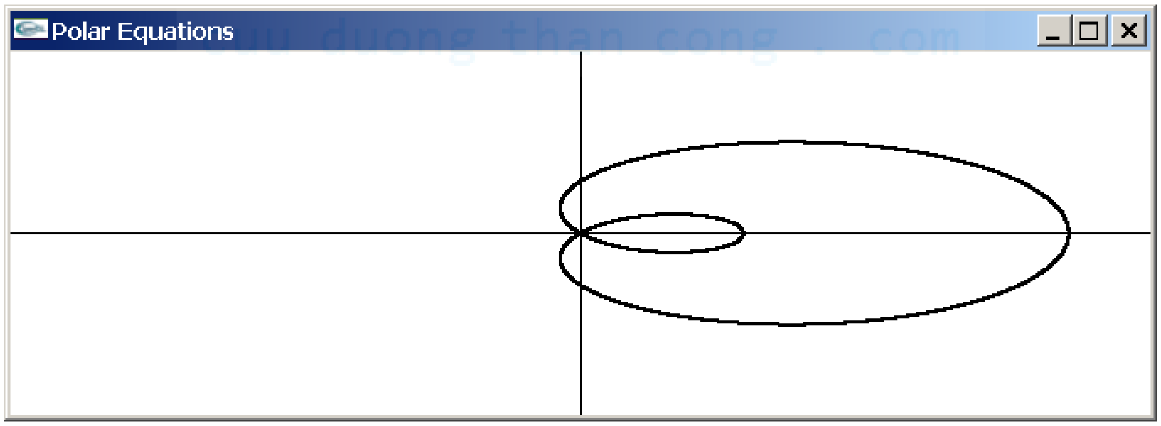


Figure 5.22

Now look at figure 5.23 WITH the **reshape** function in place.

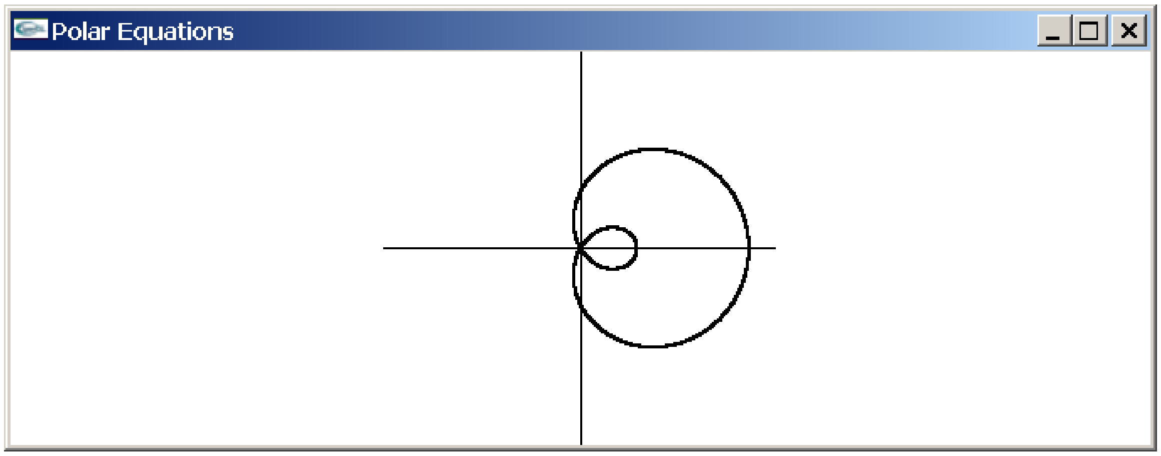


Figure 5.20

We can find fault with fact that our axis lines did not “stretch” properly (wait for the exercises!), but we can’t deny that the picture of the Limaçon now looks as it originally did in Figure 5.21, albeit a bit smaller. You may be wondering why we don’t have a **gluOrtho2D** command in the **def init():** function? Since we declared the **reshape** function as the **glutReshapeFunc()** in **def main():**, when the **pypolar.py** program runs, OpenGL/GLUT assumes that the creation of the graphics window constitutes a “reshaping” of the window and the **reshape** function is automatically called or triggered by GLUT.[11]

Also new in this program is keyboard interaction while the graphics display is visible and the program is running. We have used keyboard input before, but that input was prior to displaying a graphics window. This keyboard interactivity is a bit different. We will be able to close the program, for example, while a graphics window is open by pressing the “**Esc**” key or the “**q**” key.

```
def keyboard(key, x, y):
	#  Allows us to quit by pressing 'Esc' or 'q'
	if key == chr(27):
		sys.exit()
	if key == "q":
		sys.exit()

	# End keyboard function
```

As with the def **reshape(w, h):** function, take a look at **def main():**. You’ll see that we’ve used **glutKeyboardFunc(keyboard)** to inform OpenGL/GLUT that the **keyboard** function will provide keyboard interaction. Again, we don’t have to actually call the **keyboard** function “**keyboard**”, but any name we use for our keyboard routines must be defined in **def main** using **glutKeyboardFunc()**. You’ll notice three variable parameters in parentheses for **def keyboard**. When a “normal”[12] key is pressed, the function named in **glutKeyboardFunc** is called and the value of the key

[11] Or whatever function is name in the **glutReshapeFunc()** line in **def main():**

[12] “Normal” keys refer to letters and numbers. Other keys such as arrows or “Page Up” are called “special” keys and will be discussed later.

is sent to this “keyboard” function and stored in the variable **`key`**. In this manner, the value of the key may be processed or used by our program code. The **`x`** and **`y`** parameters are not used here. They <u>must</u> be present within the keyboard function parentheses, but we won’t discuss them at this time.

Looking closely at the **`if`** statements, if either a **`chr(27)`** value for the variable **`key`** is present (the ‘**Esc**’ key) or we press the letter **`q`**[13], Python executes a **`sys.exit()`**[14] and the program ends. Any other single key press will be ignored and the program will continue to run unless halted via the DrPython "Stop" button.

We also could have written the **`keyboard`** "**`if`**" block of code as follows:

```
if key == chr(27) or key == "q":
     sys.exit()
```

This construction basically says "if the 'Esc' key OR the 'q' key are pressed, then exit the program". This form is a bit shorter, but is it easier to understand? Perhaps. Such constructions are eventually up to the taste of the programmer.

In **`def main():`**, the only major changes are in the use of the globals **`width`** and **`height`** and the addition of the **`glutKeyboardFunc()`** and **`glutReshapeFunc()`** commands. As stated previously, the use of global variables will allow us to change the value of **`width`** and **`height`** at the beginning of the program for convenience.

Section 5.6 Conclusion

In this chapter you have been introduced to basic 2D plotting, function plotting, parametric equations, simple physics simulations, and polar equations. Hopefully you “saw” some similarities between graphing parametric and graphing polar equations as well as some differences. I hope that you are beginning to see some of the interesting possibilities available to you now that you know some programming fundamentals. In the next chapter, we'll continue with some more complex 2D graphics and create some fractal images.

Now let’s try some exercises to explore polar equations in more detail. In each exercise, see if you can determine the symmetries (if any) displayed by the graphs. Examples would be reflections in either the x or y axes (or both). Perhaps there are other symmetries as well?

[13] Only a lower case **`q`** will work… Python is very specific about this. If we want to use an upper case **`Q`**, we would have to use a function to check for either letter OR we could write another **`if`** statement and check for **`Q`**. We can use any letter as our exit key… **`q`** simply makes sense.
[14] Remember the **`import sys`** line at the beginning of the program? **`sys.exit()`** is a command provided by the **`sys`** module and stops the program immediately..

Exercises

1) In the **`def plotpolar():`** function, change the line **`r = 4*cos(theta) + 2`** to **`r = 4*sin(theta) + 2`**. What happens? **`sin()`** and **`cos()`** are exactly 90 degrees (**π`/2 radians`**)"off" or out of phase with each other. Is it surprising, then, that the graph is rotated 90 degrees? You can do similar rotations with almost all the polar equations we'll be exploring in these exercises.

Note: Plot examples for exercises 2-15 are found at the end of these exercises. Also, see the ***NOTE*** at the end of the plot examples.

2) Let's visit our old friend the Cardiod, polar equation style.[15] Change **`r = 4*sin(theta) + 2`** in exercise 1 to:

```
r = 2*a*(1 + cos(theta))
```

Assign a value of 10.0 to the **`axrng`** variable and set **`a = 2.5`**.

3) Cayley's Sextic.[16] This curve is also called a sinusoidal spiral and is formed by a cardiod rolling over another cardiod of the same size. The equation for Cayley's Sextic is:

```
r = 4*a*cos(theta/3)**3
```

An **`axrng`** of 10.0 is adequate and **`theta`** (in the **`for`** loop) should range from -6.28 to 6.28. Try 0.0 to 6.28 first and see why we need to extend **`theta`**. Try a value of 2.5 for the parameter **`a`**.

4) Cissoid of Diocles.[17] The Cissoid curve demonstrates the vertical asymptote of the tangent function. The polar equation is:

```
r = 2*a*tan(theta)*sin(theta)
```

Assign a value of 5.0 to the variable **`axrng`**. **`theta`** can range from 0.0 to 6.28. Keep the step size at 0.001 unless you want to experiment a bit.

5) One of my favorite polar equations draws a Cochleoid.[18] The cochlea is an organ found in the inner ear and it resembles a snail in that it is spiral shaped. The Cochleoid is not shaped much like a snail or a true spiral, but it is interesting nonetheless. The polar equation for the Cochleoid is:

```
r = a*sin(theta)/theta
```

[15] Kokoska, Stephen. "Fifty Famous Curves, Lots of Calculus Questions, And a Few Answers". Dept. of Mathematics, Computer Science, and Statistics, Bloomsburg University.
[16] Ibid
[17] Ibid
[18] Ibid

Here's where it gets interesting. First, assign **axrng = 1.0** and **a = 1.0**. Now let **theta** range from 0.0 to 6.28 and run the program. What happens? What caused the error? Notice that we are dividing by **theta** in the polar equation. If **theta** equals zero, we are dividing by zero… and that's not good. When we divide by a variable, we must make certain that the value of the variable does not equal zero at any time. How do we fix this? We could start theta at 0.001… something like this:

```
        for theta in arange(0.001, 6.28, 0.001):
```

But let's learn something new. When we use **from Numeric import *** at the beginning of the program, the **Numeric** module actually defines the variable **pi** for us as **3.14159265359**. This makes our task a bit easier. Try this **for** loop statement:

```
        for theta in arange(-5*pi, 5*pi, 0.001):
```

If **pi** has such a precise value (so many decimal places!) and we step by **0.001**, **theta** will never be zero. We'll "**step**" right over zero! A good question to ask at this point might be "Why did you wait so long to tell us about the **pi** variable? We could have used it instead of 3.14!" That is indeed a good question. Anyway, I like the Cochleoid. Hint: Use **glPointSize(1.0)** for a finer plot. Also, experiment a bit with the **arange** for **theta**. Try from **–pi** to **pi**. Then try from **-10.0*pi** to **10.0*pi**. How important is the loop range in **arange**?

6) Conchoid of de Sluze.[19] The Conchoid of de Sluze is a looping curve that displays x-axis symmetry. The polar equation for this curve is:

```
        r = b*cos(theta)/a - a/cos(theta)
```

We are dividing by both **a** and **theta**, so we must not allow either to be assigned a value of zero. The range for **theta** can be from **–pi** to **pi**. The **axrng** variable may be assigned a value of 1.0. Let **a = 1.0** and **b = 2.0**. Can you see the symmetry?

7) The Double Folium.[20] If my Latin is not too terrible, folium means something similar to "petal" as in a flower. So this polar equation draws a two-petal "flower". The equation is:

```
        r = 4*a*cos(theta)*sin(theta)**2
```

As in the last exercise, let theta range from **0.0** to **pi** in the **for** loop. Let **axrng = 2.0** and let **a = 1.5**. See if you can change the **theta** range in the loop and produce a "single folium".

[19] Ibid
[20] Ibid

8) Fermat's Spiral.[21] Individuals who practice omphaloskepsis have adopted this interesting curve as their symbol. You'll have to look up omphaloskepsis. Trust me, it's strange. Anyway, the polar equation is:

```
r = a*sqrt(theta)
```

Change the **`theta`** loop to look like this:

```
for theta in arange(0.0, 10*pi, 0.0001):
```

The value for **`theta`** can't be less than zero (why?). The step size has been decreased to **`0.0001`** so we can watch the plot as it is drawn. Assign **`a = 1.0`** and **`axrng = 5.0`**. For the purposes of symmetry, add an additional **`glVertex2f`** command:

```
glVertex2f(-x, -y)
```

What happens when we add this additional **`glVertex2f`**? Try the program first with just the original **`glVertex2f(x,y)`** command and then try it with both. What kind of symmetry do you see? Does the Fermat's Spiral figure at the end of the exercises use one or both **`glVertex2f`** commands? You can go back and explore some of the earlier exercises by inserting the additional **`glVertex2f(-x, -y)`** and see how the plots change. Some of the graphs are very interesting! Experiment with switching the positions of the **`x`** and **`y`** variables in **`glVertex2f`**. What happens?

9) Folium.[22] In exercise 7, you were asked to attempt the creation of a single folium. Now we'll explore the polar equation for a folium:

```
r = -b*cos(theta) + 4*a*cos(theta)*sin(theta)**2
```

First, remember to comment out (using #) the **`glVertex2f(-x, -y)`** from the last exercise. We'll use it again later, so don't erase it. Set **`axrng = 1.0`** and set parameter **`a = 0.25`** and parameter **`b = 1.0`**. Let **`theta`** range from **`0.0`** to **`pi`**. Now let's experiment a bit. Try **`a = 1.0`** for a trifolium! What happens if **`a = 0.5`**? Experiment with different values of a and b. For extra fun, uncomment the **`glVertex2f(-x, -y)`** command and see what happens. Again, consider the types of symmetry you see in the plots. When you are finished, don't forget to once again comment out **`glVertex2f(-x, -y)`** for future exercises.

10) Freeth's Nephroid.[23] This curve is a bit more kidney shaped than the nephroid we explored in the parametric equation section. Freeth's Nephroid is plotted using the following polar equation:

```
r = a*(1 + 2*sin(theta/2))
```

---

[21] Ibid
[22] Ibid
[23] Ibid

Let **`axrng = 3.0`** and **`a = 1.0`**. **`theta`** should range from **`-2*pi`** to **`2*pi`**. You can experiment a bit by dividing **`theta`** by different values. For example, try **`r = a*(1 + 2*sin(theta/3))`** and have **`theta`** range from **`-4*pi`** to **`4*pi`** in order to complete the plot. Check out the "**Mod Freeth**" figure at the end of these exercises for the **`(theta/3)`** example.

11) The Hyperbolic Spiral.[24] You may remember from your algebra class that the equation **`y = 1/x`** is the classic form for a hyperbola. We are going to create a spiral curve using a polar equation based on the classic hyperbola function. Here is the equation:

    ```
    r = a/theta
    ```

    The polar form is very simple and doesn't involve trig functions. We must be careful and not allow **`theta`** to equal zero. Try this **`for`**"\ statement:

    ```
    for theta in arange( 0.0001, 10*pi, 0.001):
    ```

    Notice how we start at 0.0001 and step by 0.001. This insures that we will never "hit" zero in our loop. Set **`axrng = 1.0`** and **`a = 1.0`**. For an interesting modification, let **`theta`** range from **`-10*pi`** to **`10*pi`** and see what happens. Finally, uncomment the **`glVertex2f(-x,-y)`** statement (or add it after the regular **`glVertex2f(x,y)`** it has been deleted). Run the program again. This spiral is similar to a curve called the Lituus[25] and is shown in the "**Lituus**" figure at the end of these exercises. Think “symmetry”!

12) Rhodonea Curves.[26] This curve is also one of my favorites. The polar equation for plotting Rhodonea curves is:

    ```
    r = a*sin(b*theta)
    ```

    Don't forget to comment the **`glVertex2f(-x,-y)`**! Let **`theta`** range from **`0.0`** to **`2*pi`**, set **`axrng = 1.0`**, **`a = 1.0`**, and **`b = 5.0`**. If **`b`** is an odd integer, then there will be **`b`** lobes (folia?) in the curve. If **`b`** is even, there will be **`2b`** lobes in the curve. If **`b`** is an irrational number, then the number of lobes will be infinite and the curve will never close on itself. We can't "do" irrational numbers in Python, but we can let **`b`** equal a decimal number. Try **`b = pi`** (π is an irrational number, but not on a computer… why?) and set **`theta`** to range from **`0.0`** to **`20*pi`**. Neat, huh? The figure "**20*pi**" illustrates the parameter **`b = pi`**.

13) The Spiral of Archimedes.[27] Perhaps the easiest polar equation of all, the Spiral of Archimedes produces, well… a spiral. The equation couldn't be simpler:

    ```
    r = a*theta
    ```

[24] Ibid
[25] Ibid
[26] Ibid
[27] Ibid

Set **`a = 0.1`**, **`axrng = 10.0`**, and the **`theta`** range from **`0.0`** to **`30*pi`**. Some variations you can try are to let **`theta`** range from -30*pi to 30*pi and uncommenting **`glVertex2f(-x,-y)`**. Don't forget to "re-comment" this statement prior to working the next exercises!

14) Butterfly Curves.[28] Butterfly curves were discovered by Temple H. Fay at the University of Southern Mississippi. They are graceful objects exhibiting symmetry and complex loops and lobes. Whether they actually resemble butterflies or moths is in the eye of the beholder, but they are undeniably interesting. The polar equation for the Butterfly Curve is:

```
        r = exp(cos(theta))-2*cos(4*theta)+sin(theta/12)**5
```

Let **`axrng = 5`** and allow **`theta`** to range from **`0.0`** to **`30*pi`**. The "Butterfly Curve 1" figure at the end of the exercises illustrates this plot. Now try this equation:

```
r = exp(sin(theta))–2*cos(4*theta)+sin((2*theta-pi)/24)**5
```

Use the same parameters as in the first butterfly curve. The results are shown in the “Butterfly Curve 2” figure at the end of these exercises. What do you think caused the 90 degree or $\pi/2$ rotation? Experiment with either or both of these equations. Change the coefficients and the range for **`theta`** and see what you can create! Butterfly curves have a unique beauty and grace, making them particularly pleasing to the eye. Research Butterfly Curves online and see what you can learn. Perhaps you can create your own Butterfly Curve?

15) Chrysanthemum Curve.[29] The Chrysanthemum Curve was also discovered by Temple Fay, so we can perhaps expect it to be somewhat similar to the Butterfly Curve. A chrysanthemum is a flower and the curve bears some resemblance to the petals on a flowering plant. The polar equation for a chrysanthemum curve is:

```
r = 5*(1+sin(11*theta/5))-(4*sin(17*theta/3)**4)*(sin(2*cos(3*theta)-28*theta)**8)
```

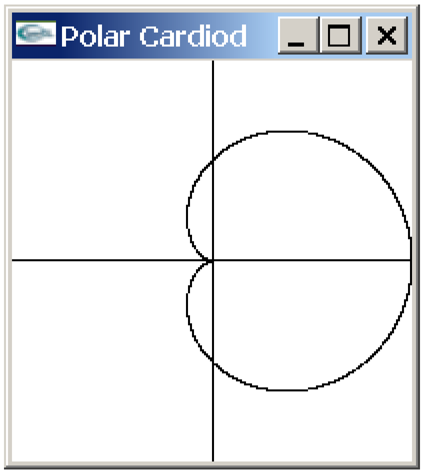
This is a long and complicated equation, so check your typing carefully! For best results, set **`axrng = 10.0`** and let **`theta`** range from **`0.0`** to **`21.0*pi`**. Feel free to experiment with the equation and see if you can create a new species of flower! Later we may revisit this polar equation when we work with 3D curves!

16) Now it’s time for you to experiment on your own. We are at the end of chapter 5 and you know enough to try something of your own creation. Use the programs in this chapter and the exercises at the end of each section as models. Try to create a “work of art” using different colors, symmetry, and different equations. Feel free to change the graphics window size and shape as well as experimenting with the background color, **`glPointSize`** and anything else you

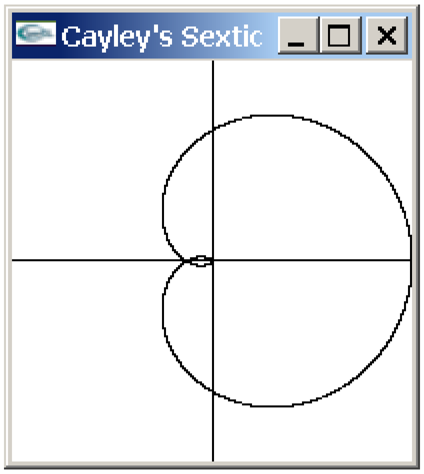
[28] Pickover, Clifford A. (1991). "Computers and the Imagination: Visual Adventures Beyond the Edge." St. Martin's Press: New York. Pages 19-21.
[29] Bourke, Paul. http://astronomy.swin.edu.au/~pbourke/curves/chrysanthemum/

can think of doing. For example, you might try to draw a “butterfly” on a “chrysanthemum”? Use your imagination… a computer is an imagination machine. What we can imagine, we can create!

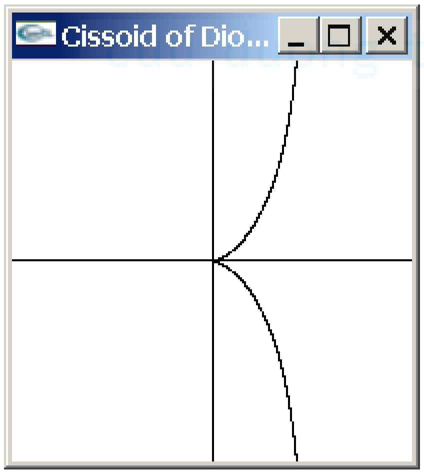
Figures for Exercises 2-15



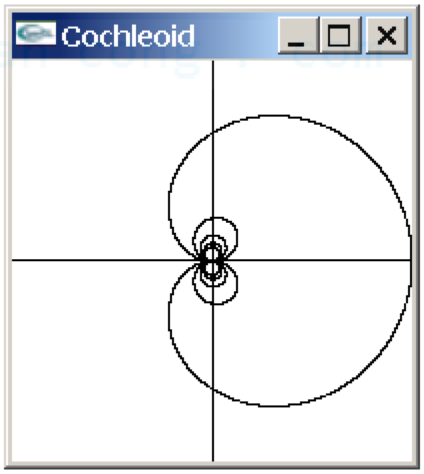
Cardiod



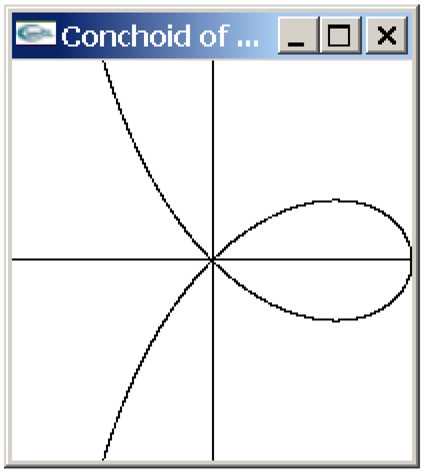
Cayley’s Sextic



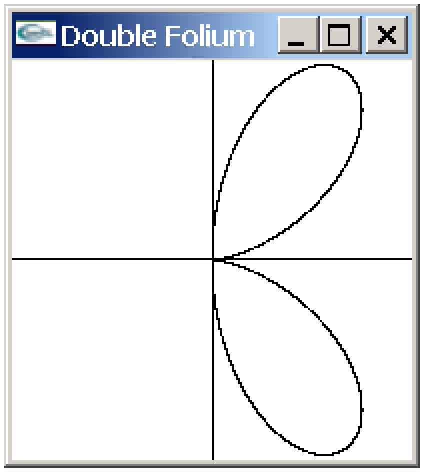
Cissoid of Diocles



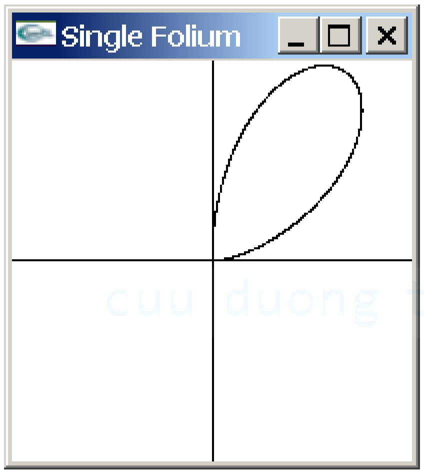
Cochleoid



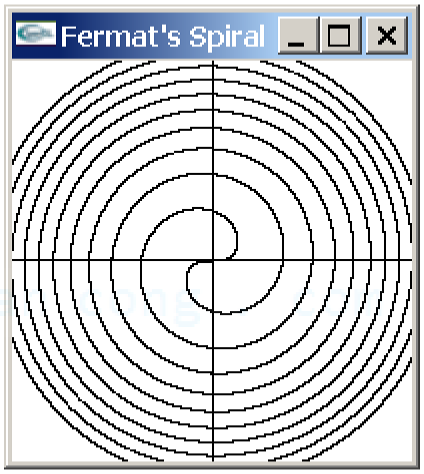
Conchoid of de Sluze



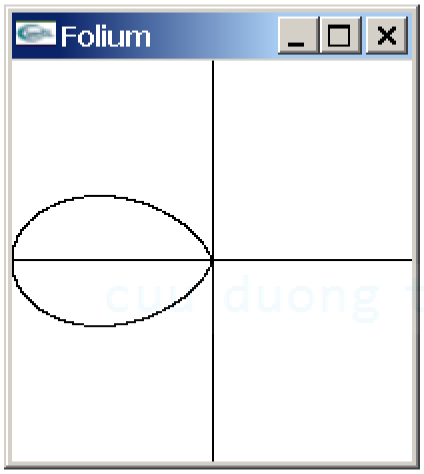
Double Folium



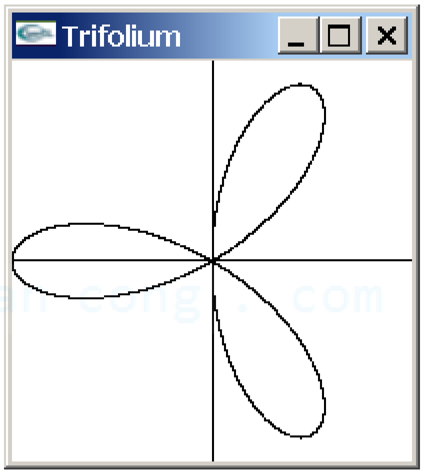
Single Folium



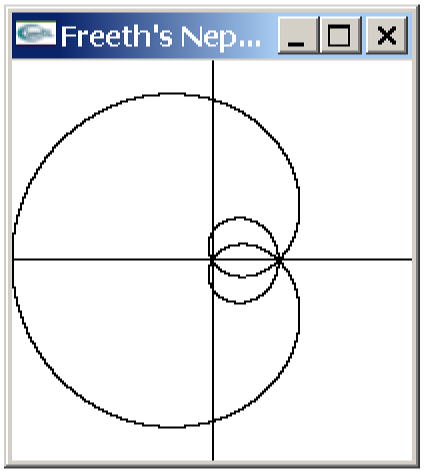
Fermat's Spiral



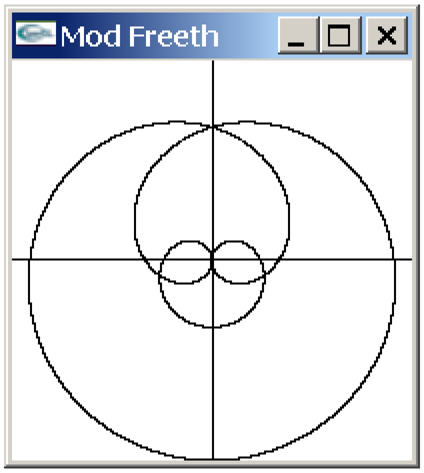
Folium



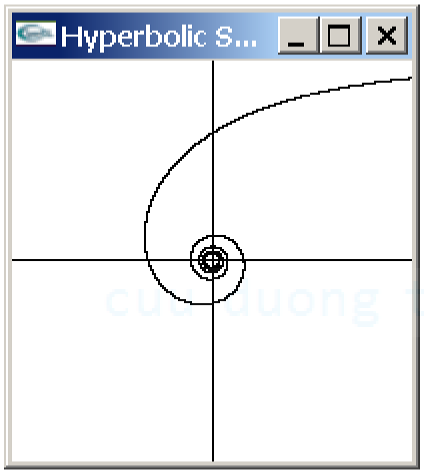
Trifolium



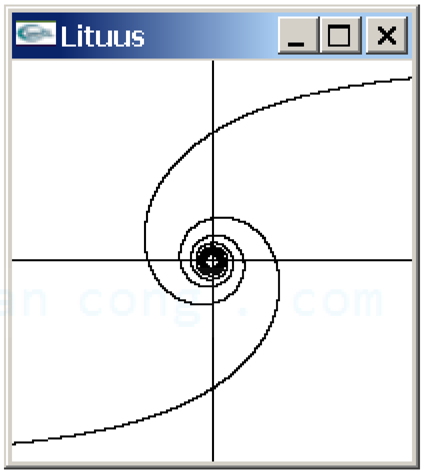
Freeth’s Nephroid



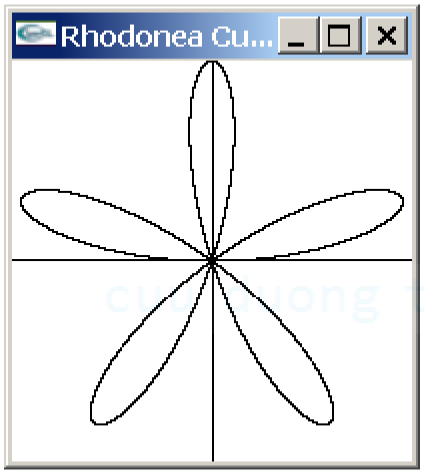
Mod Freeth



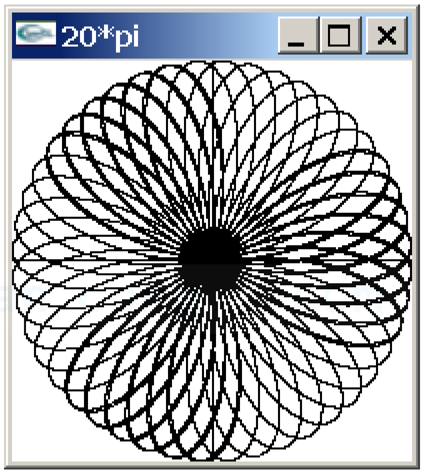
Hyperbolic Spiral



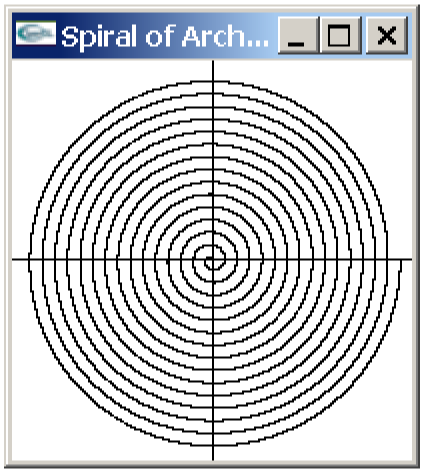
Lituus



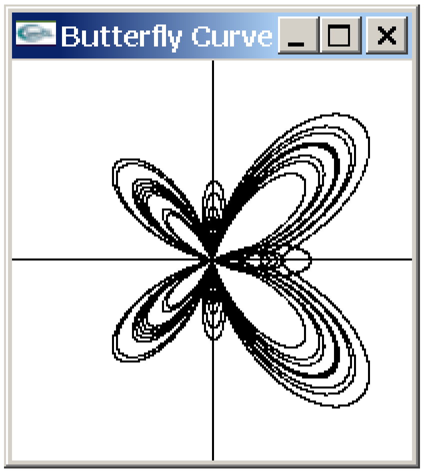
Rhodonea Curves



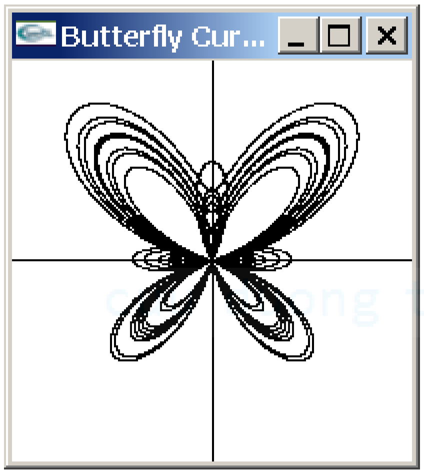
20*pi



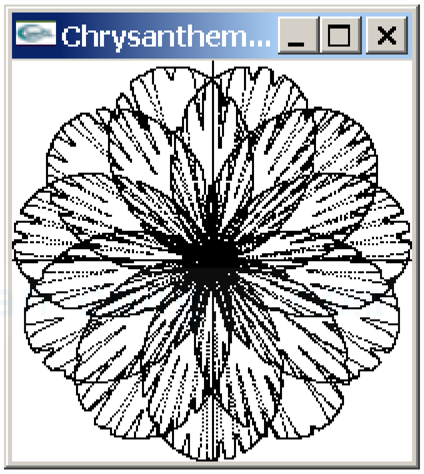
Spiral of Archimedes



Butterfly Curve 1



Butterfly Curve 2



Chrysanthemum Curve

*****NOTE***** If you are having problems creating plots that look like the examples figures on this and previous pages, make certain that you:

a) Pay attention to the suggested range for **`theta`**, the **`axrng`** value, and the values for any parameters such as **`a, b, c`**, etc.

b) Make certain that you have typed in the proper equation for **`r`**, and

c) Make certain that you have not altered the equations for **`x`** and **`y`**. They should be (for polar equations):[30]

```
x = r*cos(theta)
y = r*sin(theta)
```

[30] You CAN alter these equations, but you are no longer mapping polar equations to the standard Cartesian x-y coordinate system. The results of changing these equations can be bizarre!

Chapter 6 Patterns and Chaos in 2 Dimensions

This chapter will build on the skills we've learned in previous chapters and introduce you to some concepts that, prior to the computer age, were not available to mere mortals. One of the basic foundations of math and science is the search for patterns, both in numbers as well as in nature. Patterns imply prediction and prediction implies security. As humans, we seem to be programmed to seek patterns in our daily activities and it comes as no great surprise that we find many patterns pleasing to the eye. Indeed, a lack of patterns in behaviors, artwork, or natural events can be very disturbing! In Section 6.2, we'll explore some patterns that could easily be considered mathematical art. First, though, we need to make your programming life a bit easier.

Section 6.1 PySkel

You may be thinking that we are doing a lot of typing to display some relatively simple graphics. I might disagree with this, but I understand your viewpoint. With that in mind, let's see if we can make the programming task a bit easier for you. If you haven't already noticed, each program you have written is similar in structure. Let's build on that concept and create a skeleton program that we can use for future exercises. As we add new concepts and ideas that we want to incorporate in all our coding examples, we'll add to the skeleton program.

Open the last program you wrote from the previous chapter, click "**File**" and "**Save As**" and rename the program **pyskel.py**.[1] Then "gut" the program, leaving the following code:

```
# PySkel.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

#  Set the global width, height, and axis ranges of the window
global width
global height
global axrng

#  Initial values
width = 500
height = 500
axrng = 1.0

def init():
     glClearColor(1.0, 1.0, 1.0, 1.0)
```

[1] In tribute to Prof. George Francis's "illiSkel" programs.

```
def plotfunc():
     glClear(GL_COLOR_BUFFER_BIT)
     glBegin(GL_POINTS)

     # Plotting routines

     glEnd()
     glFlush()

def reshape(  w,  h):

     # To insure we don't have a zero height
     if h==0:
          h = 1

     #  Fill the entire graphics window!
     glViewport(0, 0, w, h)

     #  Set the projection matrix... our "view"
     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     #  Set the aspect ratio of the plot so that it
     #  Always looks "OK" and never distorted.
     if w <= h:
          gluOrtho2D(-axrng, axrng, -axrng*h/w, axrng*h/w)
     else:
          gluOrtho2D(-axrng*w/h, axrng*w/h, -axrng, axrng)

     #  Set the matrix for the object we are drawing
     glMatrixMode(GL_MODELVIEW)
     glLoadIdentity()

def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'
     if key == chr(27):
          sys.exit()
     if key == "q":
          sys.exit()

def main():
     global width
     global height

     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
     glutInitWindowPosition(100,100)
     glutInitWindowSize(width,height)
     glutCreateWindow("PySkel")
     glutReshapeFunc(reshape)
     glutDisplayFunc(plotfunc)
     glutKeyboardFunc(keyboard)
```

```
    init()
    glutMainLoop()

main()

#End of Program
```

This **pyskel.py** skeleton can serve as a basis for writing new programs. You should note that all the main ingredients we need for proper program execution are included in the code. The **`import`** statements, the **`global`** variable definitions, the **`def init():`**, **`def reshape():`**, **`def plotfunc():`**, **`def keyboard():`**, and **`def main():`** functions are ready for our use and/or modification. As a matter of fact, **pyskel.py** will run on its own, although the output won't be very interesting.[2]

Using **pyskel.py** as a skeleton or template is simple. All you do is load **pyskel.py** into DrPython or your programming editor, change the name by using the "**File**" and "**Save As**" menu item, and modify the **`def`** functions with new commands (if needed). Certain **`def`** functions, such as **`def reshape():`**, may not change at all.[3] Other functions, such as **`def plotfunc():`** may be renamed and completely overhauled. Still other functions (as yet unwritten) may be ignored and can be commented out. In any event, **pyskel.py** should save you a considerable amount of work. As we add new items such as mouse motion, zooming, menus, etc. we can add to **pyskel.py**.

Remember to keep the original pyskel.py code intact by immediately renaming the skeleton program as soon as you load it. Carefully compare the new program listing with the "old" (and hopefully renamed!) **pyskel.py** listing and make any needed changes. Such changes may include modifications to global variables such as **`axrng`** and/or **`height`** and **`width`** as well as changes in **`def`** functions and modifications to **`def main():`** to reflect new functions and/or new function names. Now let's put **pyskel.py** to work!

Section 6.2 Some Interesting Patterns

The inspiration for this particular topic comes from A.K. Dewdney's "The Armchair Universe".[4] Dewdney uses the phrase "Wallpaper for the Mind" as the title to the chapter, but I don't know much about wallpaper. Let's simply say that we are going to create some interesting patterns using our computer. I think you'll enjoy the example program and the possibilities it represents. After presenting the program listing, I'll attempt to explain how it works and the exercises will allow you to explore the mathematics of creating your own unique patterns.

[2] Try it! Before you do, though, try to guess what the output will be.
[3] **NOTE***** If your program doesn't display properly, check the **`def reshape(w,h):`** function carefully, particularly the **`glMatrixMode`** commands!
[4] Dewdney, A. K. (1988). "The Armchair Universe: An Exploration of Computer Worlds". W. H. Freeman and Company, New York. Pages 15-26.

Load **pyskel.py** and "Save As" **pymathart.py** or something similar. Change the code in the listing so that it matches the following program listing:

```
# PyMathArt.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

#  Set the global width, height, and axis ranges of the window
global width
global height
global axrng

#  Initial values
width = 500
height = 500
axrng = 10.0

def init():
     glClearColor(1.0, 1.0, 1.0, 1.0)

def plotmathart():
     glClear(GL_COLOR_BUFFER_BIT)
     glBegin(GL_POINTS)

     for x in arange(-axrng, axrng, 0.04):
          for y in arange(-axrng, axrng, 0.04):
               r = cos(x) + sin(y)
               glColor3f(cos(y*r), cos(x*y*r), sin(r*x))
               glVertex2f(x,y)
     glEnd()
     glFlush()

def reshape(  w,  h):

     # To insure we don't have a zero height
     if h==0:
          h = 1

     #  Fill the entire graphics window!
     glViewport(0, 0, w, h)

     #  Set the projection matrix... our "view"
     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     #  Set the aspect ratio of the plot so that it
     #  Always looks "OK" and never distorted.
     if w <= h:
          gluOrtho2D(-axrng, axrng, -axrng*h/w, axrng*h/w)
     else:
          gluOrtho2D(-axrng*w/h, axrng*w/h, -axrng, axrng)

     #  Set the matrix for the object we are drawing
     glMatrixMode(GL_MODELVIEW)
     glLoadIdentity()

def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'
     if key == chr(27):
          sys.exit()
     if key == "q":
          sys.exit()

def main():
     global width
     global height

     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
     glutInitWindowPosition(100,100)
     glutInitWindowSize(width,height)
     glutCreateWindow("Math Art Patterns")
     glutReshapeFunc(reshape)
     glutDisplayFunc(plotmathart)
     glutKeyboardFunc(keyboard)

     init()
     glutMainLoop()

main()

#End of Program
```

When you run this program it may take several seconds or even minutes for the entire graphics picture to display depending on the speed of your computer.[5] The reason for this slow execution is that we are "visiting" every pixel or dot within the graphics window and applying a mathematical function to each pixel. We then have the Python code decide the color for the pixel by applying another set of functions. When finished, the output for this program should look something like Figure 6.1 on the next page.

[5] Some of the graphics we'll create may take some time to complete. To put this into perspective, 20 years ago in our "old" Apple lab a graphic that now takes a minute to draw might have taken several hours or even days to finish and the completed picture would have been in glorious green and black. Hopefully any short wait time we suffer will be well worth the picture we draw!

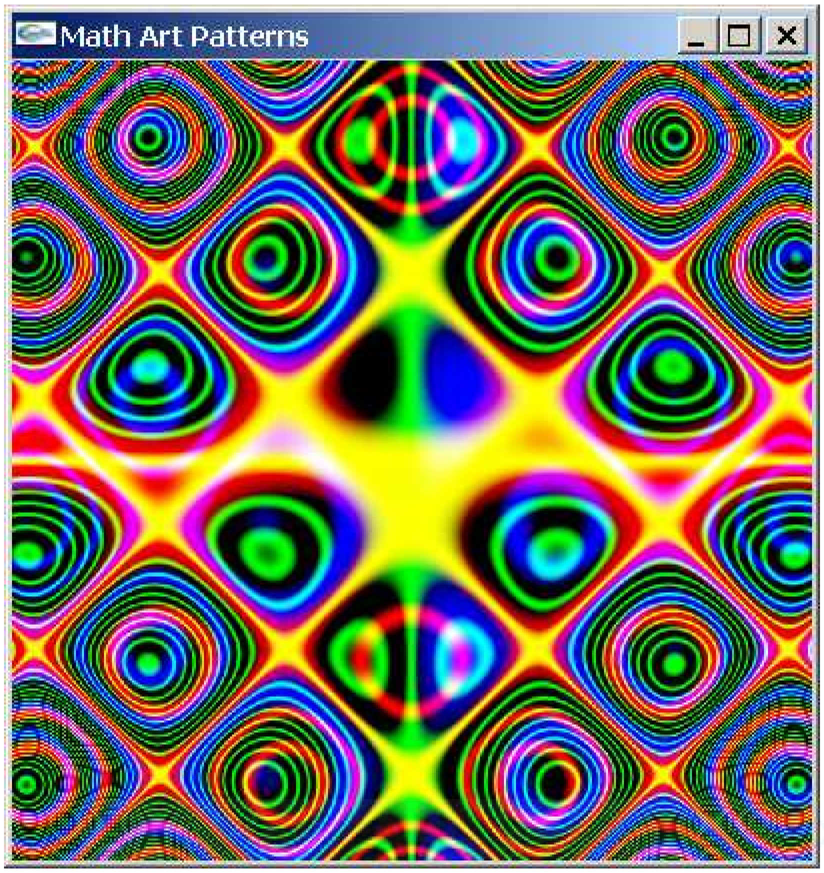


Figure 6.1

What do you think? We haven't really added much to the programs from the previous chapter and look what happened! If you look at the graphic output long enough, you can even see hints of a layered or 3D effect even though no such structure was intended. Let's see if we can understand how such interesting patterns were created. I am going to preface the explanation by saying that some of the pattern that you see is caused by an artifact introduced by the nature of the graphics screen. We can only represent anything we draw by using single pixels. This makes is virtually impossible to draw perfect straight lines and curves. There will always (almost) be some jagged edges or "jaggies" that will cause some interesting patterns in plots such as Figure 6.1. You might want to look up "**aliasing**" and "**Moire Patterns**" online?

First, we set the **`global`** variables as follows:

```
width = 500
height = 500
axrng = 10.0
```

If you look at the program listing, you'll find that the only major changes are found in the display function, named **`def plotmathart():`**, so let's look there.

```
def plotmathart():
      glClear(GL_COLOR_BUFFER_BIT)
      glBegin(GL_POINTS)
```

```
    for x in arange(-axrng, axrng, 0.04):
         for y in arange(-axrng, axrng, 0.04):
              r = cos(x) + sin(y)
              glColor3f(cos(y*r), cos(x*y*r), sin(r*x))
              glVertex2f(x,y)
    glEnd()
    glFlush()

#  End Function
```

The function begins as usual by clearing the graphics windows and specifying that we'll be plotting points by using **glBegin(GL_POINTS)**. We then encounter a nested loop structure. The "top" or "outer" loop, **for x in arange(-axrng, axrng, 0.04):** begins with the leftmost column of pixels (**-axrng**) and eventually chooses every single column of pixels from left to right until it finishes at the far right of the screen (**axrng**). The variable **x**[6] begins with **–axrng** and ends with **axrng**, stepping by **0.04**. You may be wondering why **0.04** was chosen as the step? The width of the graphics window is 500 pixels (**width = 500**) and the value for **axrng = 10.0**. This means that the screen coordinates are from **-10.0** to **10.0** in both the **x** and **y** axes, for a total width (and height) of 20.0 graphic screen units. This 20.0 unit **width** and **height** must be mapped into a 500 pixel window screen **width** and **height**. Doing a little arithmetic tells us that 20.0/500 = 0.04. So, if we step by 0.04, we'll hit every pixel column and row… in other words, every single pixel or "dot" in the window! So, according to the screen coordinates,[7] each pixel in the graphics window translates to a distance of 0.04 square units.[8]

The outer **for x** loop visits every vertical column from left to right (**-axrng** to **axrng**), but in order to complete the journey to every single pixel, we must also choose individual rows. The "inner" loop, **for y in arange(-axrng, axrng, 0.04):** accomplishes this task. The two loops work together as follows: for every value of **x** chosen (an individual column), the values of **y** range from the bottom of the window to the top of the window (**-axrng** to **axrng**), hitting each row in turn. When the **y** loop has finished its task, the next value (column) for **x** is chosen and the **y** loop repeats. You can actually see this happen on your screen. When you run this program, notice how the screen fills from left to right a column at a time. You may even be able to see the bottom to top filling? If you can't see the filling, place an additional **glFlush()** statement immediately after the **glVertex2f()** statement. What do you think this does?

[6] We don't have to use **x** as our variable… we can choose any variable we want (within reason), but since we are using the variable to store horizontal screen positions, **x** makes sense. The same reasoning holds for **y**.
[7] The coordinates of the graphics window are pixel based. A width of 500 means that the coordinates for the x axis range from 0 to 500 pixels. Using **gluOrtho2D** changes the way we address each pixel and produces the **axrng** screen Cartesian coordinate system..
[8] You may wonder what happens if we change **axrng** or the **width** and **height** of the graphics window? Good question! See exercise 1.

When we visit each pixel on the graphics screen, we must have the computer decide what to do with the selected pixel. The only raw information we have available are the **`x`** and **`y`** coordinates (in **`axrng`** or screen units). The program must somehow transform the **`x`** and **`y`** coordinate values into something new. It may not seem as if having such a small amount of information could lead to anything graphically interesting, but this is exactly how Figure 6.1 was produced. Let's see how the "magic" was done:

```
r = cos(x) + sin(y)
```

First, we use the **`x`** and **`y`** information to obtain a single value **`r`** by simply adding the **`cos(x)`** and the **`sin(y)`**. This equation is nothing special. It was simply "made up" on the spur of the moment. What we do with the value of **`r`** is where the graphic design "magic" is formed.

```
glColor3f(cos(y*r), cos(x*y*r), sin(r*x))
```

We use the value for **`r`**, along with the **`x`** and **`y`** screen coordinates, to determine the color of each individual pixel. The trig functions in **`glColor3f`** are periodic, which accounts for the swirling multicolor patterns in Figure 6.1. The **`def plotmathart():`** function finishes in a familiar fashion with **`glVertex2f(x,y)`**, **`glEnd()`**, and **`glFlush()`**.[9] The only other changes are in **`def main():`** to indicate the graphic window caption in **`glutCreateWindow`** and the name of the display function in **`glutDisplayFunc.`**

In summary, we did the following:

1. Used nested loops to visit each pixel in the graphics window.
2. Used the screen (**`axrng`**) **`x`** and **`y`** coordinates in a mathematical function to obtain a value, which we stored in the variable r.
3. Used the variable **`r`**, along with the **`x`** and **`y`** coordinate values, to produce a pixel color.

We'll use these steps and this concept again. When you think about it, producing any graphics in a window involves working with individual pixels and changing their colors!

Exercises

1) We used a step of 0.04 to visit every pixel in our graphics window. The 0.04 step size was calculated based on the specific **`width`**, **`height`**, and **`axrng`** of this particular example. If we change the **`width`**, **`height`**, and/or **`axrng`**, we'll need to recalculate the step size. To see why, set **`width = 600`** and **`height = 600`** and run the program again.[10] What did you see? How did the white lines get there?

[9] Try running the program by commenting **`glFlush()`** with a # sign. Does it work? Are there any problems when you resize or move the graphics window?

[10] We keep **`width = height`** for a square window. This is "forced" on us by the **`reshape`** function. You can set the **`width`** and **`height`** to different values, but only a square area will plot.

Reset the **`width`** and **`height`** back to **`500`** and set **`axrng = 20.0`**. Run the program again.[11] This time we did not get white lines even though we increased the screen range from -20.0 to 20.0. But is the step size correct? Perhaps we are doing too much work and using a step size that is too small? This will not produce an incorrect plot, but it will increase the time it takes to complete the graphic display. Set the step size in both loops to 0.08 and run the program again. The execution time is much faster and the display is the same. It turns out that 40.0/500 = .08.[12] OK, now for the question. Is there a way for Python to automatically calculate the step size based on the values for **`width`**, **`height`**, and **`axrng`**? How might you do this?

2) If you were stumped by exercise 1, here are a couple of hints. We might first ask the question "How did we calculate the ideal step size manually?" Then we could break down the manual solution into steps and convert each step into Python code.[13] Many problems can be solved in this manner.[14] See if you can modify the **pymathart.py** program to automatically calculate the step size. How would you know if it works properly?

3) Here's a possible solution to calculating the step size automatically. First, add the following global variable underneath the **`global width`** and **`global height`** variables at the beginning of the program:

```
global stepsize
```

Then underneath the "**`axrng =`**" assignment statement, add the following calculation:

```
stepsize = (2.0*axrng)/width
```

We multiply **`axrng`** by 2.0 because **`axrng`** takes on all values from **`-axrng`** to **`+axrng`**, which is exactly twice the value of **`axrng`**. We divide by **`width`** since the **`width`** and **`height`** are equal and represent the number of horizontal and vertical pixels in the graphic window. The result is the proper stepsize for these values.

Finally, in both **`for`** loops, change the step value (probably 0.04) to the variable **`stepsize`** as follows:

```
for x in arange(-axrng, axrng, stepsize):
     for y in arange(-axrng, axrng, stepsize):
```

[11] This causes us to have a "zoomed out" view of the plot.
[12] 40.0 is the total **`axrng`** from -20.0 to 20.0.
[13] Of course, Python isn't the only language! Some programming languages are better than others at a particular task. The "J" language is excellent for mathematical and matrix programming, for example. Ruby is outstanding for web programming through Rails.
[14] No claims are made for this method being the best or most elegant method for solving all problems with a computer. It seems to work in most cases, though, and such an approach is generally understandable. There are computer wizards who seem to be able to solve problems better than most mortals. I don't understand the ways of such programming gurus and as Tolkien said, "Do not meddle in the affairs of wizards…"

Now run the program with **`axrng = 10.0`** to verify that you can plot the same graphic as in Figure 6.1.[15] Now try **`axrng = 5.0`** followed by **`axrng = 15.0`**. Also try various values for **`width`** and **`height`** (keeping both variables equal to the same value for a square plot window).[16] Personally, I prefer **`axrng  = 15.0`**, but you are free to disagree!

4) Now let's experiment with the **`r`** equation and see what we can discover. Let **`axrng = 10.0`** and alter the **`r`** equation as follows:

```
r = cos(x) + sin(y) - tan(x*y)
```

What do you think? Now try:

```
r = cos(x**2) + sin(3*y) - tan(x*y/4)
```

Weird!? Try your own equations. Experiment! I'm certain you can do better than I can at creating your own artistic patterns! You can also use **`sqrt()`**, **`log()`**, and **`exp()`** as functions. If you generate an error message and the program doesn't run, more than likely you are supplying a function with a value that causes that function to be undefined, such as the square root or log of a negative number or trying to take the log of zero. One way to fix such problems is to use the **`abs()`**[17] function as follows:

```
r = cos(x)**2 + sin(y/3) – sqrt(abs(x*y))
```

Note the use of **`abs(x*y)`** within the **`sqrt`** function to insure that **`x*y`** is never a negative number.

5) What if we want to use a function such as **`log(r)`**? The variable **`r`** can never be zero because **`log(0)`** is undefined. We can fix this problem by using an **`if`** statement. Try this **`r`** equation without any other program modifications:

```
r = cos(x)**2 + sin(y/3) - log(abs(x*y))
```

The program acts as if it's going to run and then it stops. If you look at the console window below the code, you'll see that Python "flags" this statement and tells you that we've generated an "**`OverflowError:  math range error`**". Why? Because at some point **`abs(x*y)`** within the **`log()`** function equals zero! We can fix this problem by either making certain that **`abs(x*y)`** is never zero or (better) we can simply ignore or skip the **`r`** equation if **`abs(x*y)`** equals zero.

[15] You must use 10.0 or some other floating point value that displays a decimal. Otherwise the division in the **`stepsize = (2*axrng)/width`** will result in a value of zero (an integer).
[16] You might change the "**`height =`**" statement to **`height = width`**. That way you only have to change one number to adjust the size of the graphics window.
[17] Absolute value. The log of zero is a bit more problematic. One solution would be to simply make certain the variable or statement within the **`log()`** function is never zero. The best solution is to use an **`if`** statement so that you only execute the **`log()`** function when the statement is not zero. See exercise 5!

Here's how. After the **`for y`** loop (shown), modify the code as follows:

```
for y in arange(-axrng, axrng, stepsize):

        if x <> 0:
                r = cos(x)**2 + sin(y/3) - log(abs(x))
```

Note the indent in the **`r`** equation after the **`if`** statement.[18] An English translation of this code is as follows: IF the value for **`x`** is NOT equal to zero (we used "**`!=`**" for NOT equal in the Super-3 program. If you will recall, I told you then that "**<>**" also means NOT equal), then **e**xecute the **`r`** equation. This is exactly the behavior we want! We want to have Python evaluate an equation only if the equation is "legal". If **`x`** IS zero, then **`log(abs(x))`** is NOT defined. The **`if`** statement "catches" this problem and skips the **`r`** equation when **`x = 0`**. Now if you run the program, you should be able to generate a graphics plot. The drawback to using an **`if`** statement in this manner is that, depending on which math functions you use, you may have to modify the conditional every time you change the equation to avoid encountering an undefined value. This usually isn't difficult, though.[19]

6) Try this slightly modified equation for **`r`**:

```
r = cos(x/y)**2 + sin(y/3) - log(abs(x))
```

When you run the program, it doesn't work! Even with the **`if`** statement to avoid **`x == 0`**, we still get an error. Can you see why the error is generated? Look at the **cos(x/y)** command. What happens if **`y = 0`**? We can fix this problem by using a more complex **`if`** statement as follows:

```
if x <> 0 and y <> 0:
        r = cos(x/y)**2 + sin(y/3) - log(abs(x))
```

This conditional block of code executes <u>only</u> if <u>both</u> **`x`** AND **`y`** are NOT equal to zero.[20] Run the program again. This time it should work and you may see an interesting pattern!

7) Now let's turn our attention to the line of code that determines the color of each pixel. We'll find out that this code statement is no less important than the equation for **`r`** in determining the pattern plotted in the graphics window! First, let's return everything back to its original state so that we can once again plot a pattern resembling Figure 6.1. You can refer to the **`global`** variables and **`def plotmathart():`** code listed immediately after Figure 6.1 if needed. Run the program to make certain you can plot Figure 6.1 again.

Let's modify the **`glColor3f`** statement as follows:

[18] Any and all lines indented at this level after an **`if`** statement are included in the **`if`** block.
[19] You might have to think a bit? Ouch.
[20] Another possibility is **`if x <> 0 or y <> 0:`** Would using the **`or`** logic work properly in this situation? Why or why not?

```
glColor3f(cos(r), cos(r), cos(r))
```

After running the program, you should see something like the Exercise 7 plot at the end of these exercises. What happened to all the colors, swirls, curves, and 3D effects? The **`cos(r)`** function varies according to the value of **`r`**, but since all 3 colors[21] are varying at the same time and producing the same values, the effect is to produce shades of white, gray, and black. Apparently we need to make certain that the red, green, and blue color values vary at different rates or by different functions in order to create colors and patterns like we saw in Figure 6.1.

8) Building on the previous exercise, let's explore some "color" functions. Try this:

```
glColor3f(r*r, x*r, y*x*r)
```

Sadly, the plot looks like something from the '60's… we used to call it "modern art". Unfortunately it's no longer "modern" and whether it was ever "art" is certainly open for debate. However, you probably noticed that the underlying pattern was similar to the Exercise 7 plot. Evidently the intrinsic pattern is produced by the trig functions in the **`r`** equation,[22] but both the **`r`** equation and **`glColor3f`** statements work together to produce the intricate designs and colors found in Figure 6.1.[23] An illustration of this plot is found in the Exercise 8 figure at the end of these exercises.

9) Keeping the **`r`** equation the same as in the original program, modify the **`glColor3f`** command as follows:

```
glColor3f(tan(r*r), sin(x*r)*cos(r*y), sin(y*x)*tan(r))
```

This plot is much better (in my opinion) than the previous example in exercise 8. Feel free to try other math expressions such as **`log()`**, **`sqrt()`**, and **`exp()`**. If you get an error message, remember that you might be trying to supply the math function(s) with an illegal or undefined value. You know how to fix that problem, don't you? The Exercise 9 plot at the end of these exercises demonstrates this **`glColor3f`** statement.

10) One more example and I'll let you explore on your own. Set **`axrng = 5.0`** and modify the **`r`** equation as follows:

```
r = cos(x)**2 + sin(y*y)**2 + tan(x*y)**2
```

Then change the **`glColor3f`** command to:

```
glColor3f(x*y*sin(r), sin(x*r*y)*cos(r*y), sin(y*x)*cos(r))
```

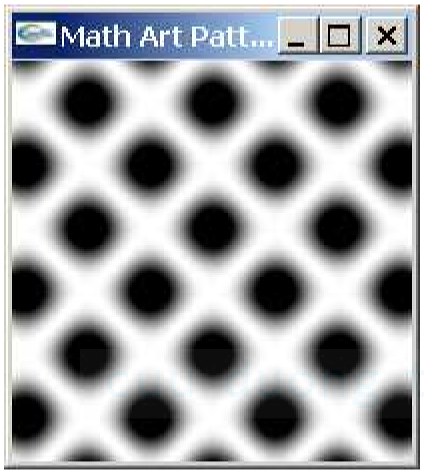
The plot generated by these statements is a bit different than the previous patterns. Do you see any symmetry? If so, what forms of symmetry did you discover? Pattern

[21] **`glColor3f(red, green, blue)`**
[22] There aren't any trig functions in this **`glColor3f`** statement!
[23] Also aliasing and "jaggies".

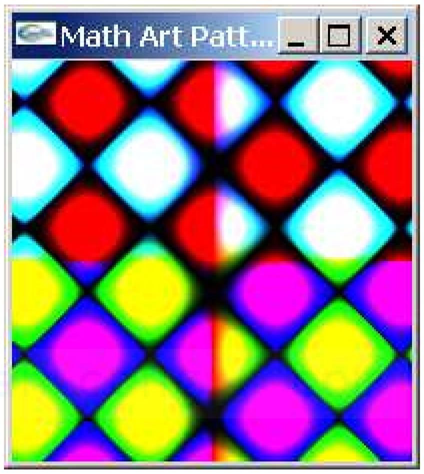
symmetry? Color symmetry? Both? Again, an example of this plot is found in the Exercise 10 Figure at the end of these exercises.

11) Now go ahead and try to create some patterns of your own. When you do your computer experiments, see if you can discover the symmetries in the plot.[24] Some of the symmetries may be subtle and could involve both pattern and color. What math functions result in symmetry and which functions cause the symmetry to break down? Do you notice any 3D effects? What do you think causes the illusion of depth?

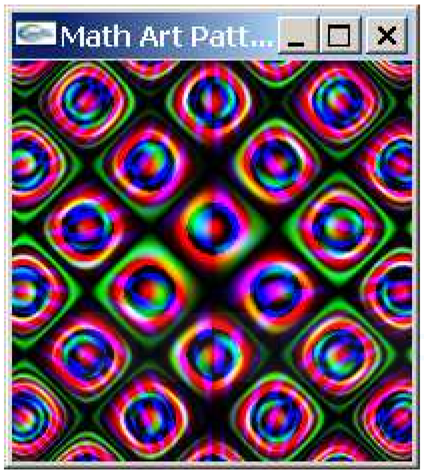
Figures for exercises 7, 8, 9, and 10



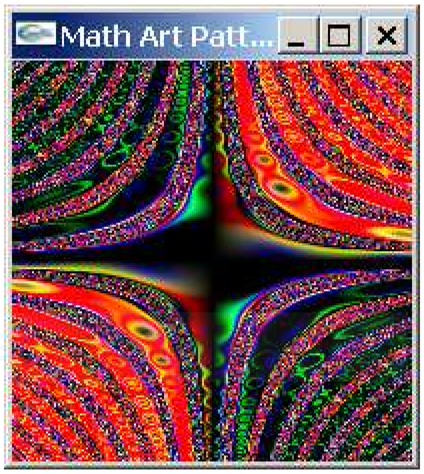
Exercise 7



Exercise 8



Exercise 9



Exercise 10

[24] If there is a lack of symmetry, do you find the pattern disturbing in any way?

Section 6.3   The Chaos Game

We aren’t really going to play a game in the sense that there will be a winner and a loser, but we are going to simulate some aspects of a game using chance and rules. But what is chance? We usually use the word "random" to describe a chance occurrence, but what do we mean by random? One possible definition is that random means unexpected. This definition is not very valid because there are many unexpected events that are not random. Did you ever receive a card, email, or letter that you didn't expect? Assuming that the message was from someone in your family or from a friend, it was hardly a random event. This individual purposely sent you the message. We might also define random as "unpredictable". It is true that random events are unpredictable, but unpredictable events are not necessarily random. As an example, I might ask "Who will win the World Series next year?" I don't know and neither do you. As I write this, my favorite baseball team, the St. Louis Cardinals, just finished winning the 2006 World Series. I could not have predicted this outcome with any certainty even a month ago. Yet the championship was not simply a random occurrence. Unpredictable and unexpected certainly, but it was not random. A multitude of sequential events (hits, runs, errors, outs, managerial decisions, wins, losses, etc.) and parallel events (other teams winning or losing) led to the final outcome.

For the purposes of this chapter, what do we mean by a random event? We must first specify all the possible events that can occur. Once we have specified all possible events that can take place, we then supply the condition that each of the possible events has an equal probability or chance of occurring. Furthermore, the occurrence of any single event is not affected by past events, nor will its occurrence affect future events.[1] Once we establish these rules, we can then say that an event is random and therefore we can now also say the event is "unpredictable". An example of a random event would be the toss or roll a fair die. There are six possible events that can occur; one event corresponding to each face or number on the die. Each of the six faces or numbers has an equal chance or probability of being "thrown". The occurrence of a "3" on one toss of the die, for example, was not based on any previous number being thrown, nor will this occurrence of a "3" have any influence on the next roll or toss of the die.

While we can't predict the next roll of the die, we can use the mathematical laws of probability to predict that over a large number of rolls (millions, billions, trillions!), we would expect that each number on the die would appear with nearly equal frequency. This definition of random is very compressed and certainly not complete enough for a mathematician, but it should serve our purpose here. The question now becomes, "How can Python generate a random number?" The short answer is that it can't. Computer languages are very much deterministic. In other words, computer languages are based on algorithms (code recipes) and formulas. We can, however, simulate random numbers in code by using rather complex formulas. Such formulas are inherent in the Python **`random`** module and the numbers they generate are called "psuedo-random"

[1] Such events are called independent events in statistical jargon.

numbers. For all practical purposes, we can treat such pseudo-random numbers as random.[2]

The Chaos Game[3] involves rolling a die and applying a rule based on the random outcome of the die. Imagine that you have placed 3 points on a large poster board and these 3 points determine the vertices of a triangle. The points can form any triangle shape, but let's assume that the triangle is at least isosceles if not equilateral.[4] Then, completely at random, choose any new point (call it P_1) on the poster board, either inside or outside the triangle. So, at this point we have the vertices of a triangle plotted on our imaginary poster board and we have a fourth point P_1 randomly drawn in another location on the poster board.

Now comes the interesting part. Label the vertices of the triangle 1, 2, and 3 respectively. As we roll the die, let an outcome of 1 or 2 "choose" or map vertex 1. An outcome of 3 or 4 will map vertex 2, and an outcome of 5 or 6 will map vertex 3. Since this is an imaginary exercise (at this point), we can even use an imaginary 3 sided die, with each side corresponding to one of the 3 vertices.[5] The rule we apply is as follows: starting with the first random (non-vertex) point P_1, role the die and plot the midpoint between P_1 and the vertex chosen by the die. Label this point P_2. P_2 now becomes the new starting point. Roll the die again and choose a new (or same… this is random!) vertex based on the outcome of the roll. Plot the midpoint from P_2 to the new vertex and label this point P_3. P_3 becomes the new starting point. Roll the die again to choose the next vertex. Once again, plot the midpoint between P_3 and the new vertex and label this point P_4. This process continues until you become bored or the universe ends.

Figure 6.2 illustrates the first few moves of the Chaos Game. You can probably follow along with the sample game progress.[6] Starting with point P_1, the die was rolled and vertex 1 was selected.[7] The midpoint between P_1 and vertex 1 was plotted and labeled P_2. Which vertex was chosen next by the die? That's correct, vertex 2. Notice that P_3 is the midpoint between P_2 and vertex 2. Then vertex 3 was chosen and P_4 was plotted as the midpoint between P_3 and vertex 3. Vertex 3 was "rolled" again, and P_5, the midpoint between P_4 and vertex 3, was plotted.

Here's the important question. What happens if you continue this game for 100 rolls of the dice? How about 1000 rolls? How about 10000? How about an infinite number of rolls? Will we see anything interesting or will the poster board fill with a random jumble of dots? I don't have time to do this as a "real" exercise and neither do you, but our computer can be a willing lab assistant and help us explore this problem as a virtual exercise. The trick is to translate the game instructions into Python code. The listing following Figure 6.2 is one possible solution. Go ahead and carefully type in the code now, paying close attention to spelling, punctuation (where needed), and

[2] Look up random number generators online. See if you see the phrase "linear congruential" anywhere. Also look up "Wolfram Rule 30".
[3] Barnsley, Michael (1989). "Fractals Everywhere". Boston: Academic Press.
[4] This is merely for convenience and appearance. We don't HAVE to choose vertices that form an equilateral triangle.
[5] A 3-sided die makes our imaginary game a bit easier. However, there can be no actual 3-sided die. Why? If we do this as a "real" exercise, you can use a conventional 6-sided die as described in the text.
[6] Follow the arrows!
[7] Either by a "1" on our imaginary die or a 1 or 2 on a "real" die.

indentation.[8] I strongly recommend that you use the **pyskel.py** program template we created in section 6.3 to reduce your workload. Load and then immediately save **pyskel.py** (using “File" and "Save As”) as **pychaosgame.py**.[9] Don't worry if you don't understand all of the code at this point. We'll go through the new concepts in detail after we get the program running correctly.

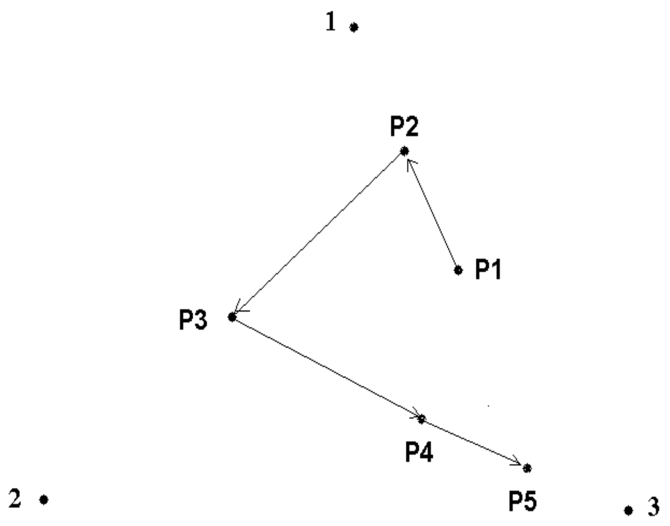


Figure 6.2

```
# PyChaosGame.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

#  Set the global width, height, and axis ranges of the window
global width
global height
global axrng
```

[8] I know you know this by now, but it doesn’t hurt to mention it again. Most Python programming errors are simple mistakes in spelling and/or indentation.
[9] You know this already, but by immediately saving the **pyskel.py** program under a new name, we protect the **pyskel.py** template program from harm. Computer savvy students can find **pyskel.py** using the File Explorer program (right-click “Start” and select “Explore”) and can change the attributes of **pyskel.py** to “Read-only” so it can’t be erased or overwritten by accident. To do this, right-click the **pyskel.py** Python icon, select “Properties” and check the “Read-only” box at the bottom of the dialog window. Linux users can do something similar using either a command line chmod statement or by using the Linux file explorer equivalent.

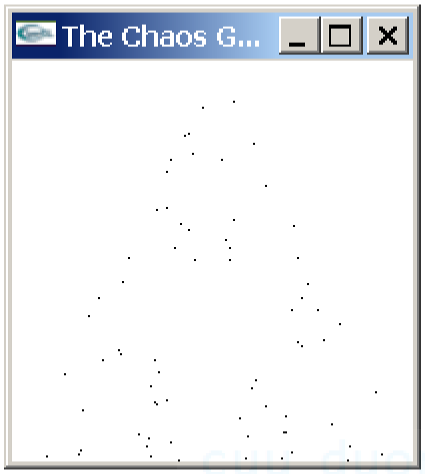
```
# Initial values
width = 500
height = 500
axrng = 2.0

def init():
     # White background
     glClearColor(1.0, 1.0, 1.0, 0.0)

     # Black Plot
     glColor3f(0.0, 0.0, 0.0)

def plotfunc():
     # Store the triangle vertices in an array
     verts = [[0.0,2.0],[-2.0,-2.0],[2.0,-2.0]]

     # Choose an initial point... any point
     x = -1.5
     y = 0.75

     glClear(GL_COLOR_BUFFER_BIT)
     glBegin(GL_POINTS)

     for n in range(0,100):
          v = randint(0,2)
          x = (x + verts[v][0])/2
          y = (y + verts[v][1])/2

          if n > 30:
               glVertex2f(x,y)
     glEnd()
     glFlush()

def reshape(  w,  h):

     # To insure we don't have a zero height
     if h==0:
          h = 1

     #  Fill the entire graphics window!
     glViewport(0, 0, w, h)

     #  Set the projection matrix... our "view"
     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     #  Set the aspect ratio of the plot so that it
     #  Always looks "OK" and never distorted.
     if w <= h:
          gluOrtho2D(-axrng, axrng, -axrng*h/w, axrng*h/w)
     else:
          gluOrtho2D(-axrng*w/h, axrng*w/h, -axrng, axrng)

     #  Set the matrix for the object we are drawing
     glMatrixMode(GL_MODELVIEW)
     glLoadIdentity()

def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'
     if key == chr(27):
          sys.exit()
     if key == "q":
          sys.exit()

def main():
     global width
     global height

     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
     glutInitWindowPosition(100,100)
     glutInitWindowSize(width,height)
     glutCreateWindow("The Chaos Game")
     glutReshapeFunc(reshape)
     glutDisplayFunc(plotfunc)
     glutKeyboardFunc(keyboard)

     init()
     glutMainLoop()

main()

# End of program
```

When you have typed the code listing and have checked for errors, run the program. Not very impressive, is it? Let's fix that! In the function **`def plotfunc():`**, change the **`100`** in **`for n in range(0,100):`**[10] to **`1000`** and run the program again. Do you see some structure? Change the **`1000`** to **`10000`** and try again. Better? Now try **`100000`** points. The structure you see is called the Sierpinski Gasket[11] and it is an *attractor*, meaning that all points after the first few are attracted to this triangular shape. We could plot millions[12] of points and they would all find their way to the Sierpinski Gasket! I find this fact to be as remarkable as it is mysterious! How can it be that a random process such as the selection of a vertex, and a rule consisting of finding the

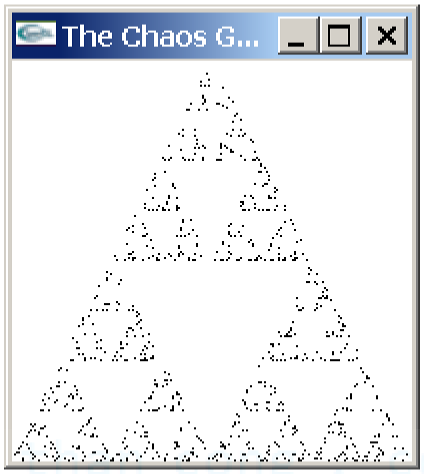
[10] You may be wondering why we used **`range(0,100)`** instead of **`arange(0,100)`**? The **`arange()`** command requires the use of **`from Numeric import *`** and **`range()`** does not. So what? Well, **`arange()`** allows us to step by decimal values such as **`0.001`** while **`range()`** is restricted to integer steps. In this program, we use integer steps and **`arange()`** is not required.
[11] Sometimes called the Sierpinski Triangle. This shape is a fractal!
[12] Actually, an infinite number of points would behave in the same manner, but would take a LONG time... well, an infinite amount of time to plot. We'll settle for thousands or perhaps millions of points in our work in this text.

iterated midpoint between the old midpoint and the randomly selected vertex, results in such a remarkable picture? This is nothing short of amazing!

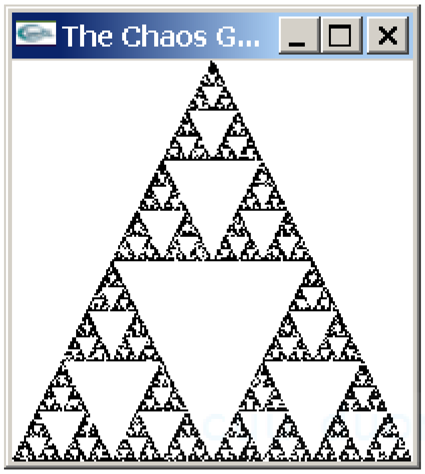
One special property of the Sierpinski Gasket is that it has no surface area. You may find this difficult to believe, but consider that we are actually plotting geometric points[13] and geometric points have zero area. The Sierpinski Gasket can be constructed in other ways and you'll have the opportunity to research this in the exercises. Example plots of the Sierpinski Gasket are found in figures Sierpinski 100, Sierpinski 1000, Sierpinski 10000, and Sierpinski 100000 below.[14]



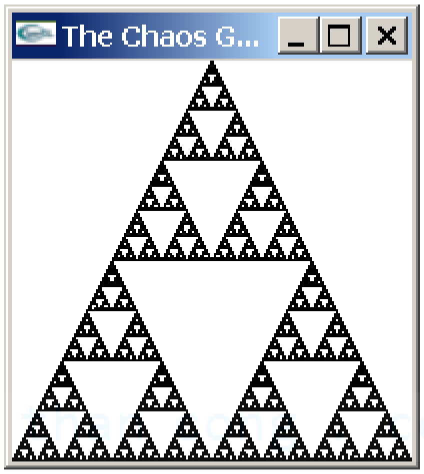
Sierpinski 100



Sierpinski 1000



Sierpinski 10000



Sierpinski 100000

What is the first thing you noticed about the Sierpinski Gasket? Triangles within triangles, of course! A second special property of the Sierpinski Gasket is that it is an example of a fractal. Fractals such as the Sierpinski Gasket exhibit self-similarity

[13] Geometric points are represented by pixels… a pixel is NOT a geometric point!
[14] The numbers refer to the number of points plotted in each figure.

(among other things) and no matter how much you magnify them, you will always encounter the same image… at least until you reach the limit of your computer's ability to distinguish one number from another.[15] For example, if you choose one of the smaller triangles within the large triangle and magnify it, you will see that the magnified image is indistinguishable from the original Sierpinski Gasket. As a matter of fact, they are exactly the same! There are as many points within ANY of the small triangles (no matter how small!) and the entire Sierpinski Gasket. This is absolutely true, but very difficult to believe. For a physical analogy, think of a hologram. A complete hologram contains enough information to display a true 3D image of an item or scene. If you cut a hologram into tiny pieces, each piece still contains the entire image, albeit from slightly different perspectives.[16] This isn't a mathematical proof, of course, but the idea can be extended to fractals in a somewhat similar fashion. Another interesting property of a fractal is that its dimension is not an integer. The Sierpinski Gasket is not a 2-dimensional object, nor is it a 1-dimensional object. Instead, its dimension is somewhere in between. A fractal has a fractional dimension.

If you want a more mathematical treatment of fractals than the hologram analogy, remember that there are as many counting numbers as even numbers and as many even numbers as odd numbers and as many prime numbers as counting numbers… all these sets contain an infinity of numbers. Likewise, there are an infinite number of points in every portion of a Sierpinski Gasket. You could magnify a small corner of the Gasket $10^{1000000}$ times and that tiny portion still contains an infinite number of points… just like the original image! We'll revisit the concept of fractals and iteration several times throughout this text. Now back to the **`pychaosgame.py`** program and the Sierpinski Gasket.

If you were paying attention to the code as you typed, you may have noticed the unfamiliar use of the variable **`verts`** in the **`def plotfunc():`** function. This construction is known as a list (in Python) or generically as an array. Arrays are a very powerful feature in any programming language and we need to discuss how they are created, accessed, and manipulated in a program.

Imagine for a moment that we need to create and store a list of 100 (x, y) coordinates. How would you do this? Well, we could create 100 "x" variables and 100 "y" variables and name them x0, y0, x1, y1, x2, y2, … x99, y99. That would be a royal pain, though, and we wouldn't have an easy way to reference or use any of the variables when we needed them. There must be a better way and that's where the concept of an array enters the scene. An array is a list of variables, all having the same variable name, but with each position in the array assigned a different number or index. In our 100 (x, y) coordinate example, we could create an "x" array and a "y" array, each capable of storing 100 (or more) values. The first "x" coordinate would be called **`x[0]`** and the first "y" coordinate would be called **`y[0]`**. The 50th (x, y) coordinate would be

[15] Contrary to popular belief, computers are NOT that great when it comes to mathematics. The real number line is a good example. You already know that the real number line is complete, meaning that there are no holes in the number line anywhere. Not so with the computer! Most computers are capable of 15 or 16 digits of precision (decimals), which means that if two values differ in the 17th decimal place or beyond, your computer can't tell the difference between them without special software.
[16] I've seen and done this… it's remarkable!

(**x[49]**, **y[49]**).[17] The value of such a construction is that we can easily run through all 100 coordinates by using a loop as follows:

```
for n in range(0,100):
     glVertex2f(x[n], y[n])
```

This small sample of code would have plotted all ordered pairs (100 of them from **x[0]**, **y[0]** to **x[99]**, **y[99]**) stored in the **x[]** and **y[]** arrays.[18] This is FAR easier than trying to create and use 2*100 individually named variables!

Arrays are an efficient method for storing hundreds, thousands, and even millions of data points. However, they can be difficult to understand, especially for a beginning programmer. Sometimes visualizing arrays as lists seems to help in the understanding of how to store and access data in an array. Let's use the x, y coordinate data we've previously discussed as an example. Imagine that the x and y data are stored in the following manner:

x[ ] variable	**y[ ] variable**
x[0]	**y[0]**
x[1]	**y[1]**
x[2]	**y[2]**
x[3]	**y[3]**
.	.
.	.
.	.
x[99]	**y[99]**

The **x[]** and **y[]** arrays are examples of single dimensional arrays. You can imagine each as a single column or list of values with each **x[n]** and **y[n]** entry capable of storing one value. We are not limited to arrays of a single column or dimension, though. We can just as easily create 2, 3, 4 or higher dimensional arrays. Arrays of 2 dimensions can be visualized as rows and columns similar to a spreadsheet. 3-dimensional arrays can be visualized as layers of spreadsheets or stacks of row and column data. Higher dimensions are a bit more difficult to visualize.

The **pychaosgame.py** code in this section uses a 2-dimensional array to store the vertex coordinates of the initial triangle shape. We'll use this program to illustrate the concept of storing and accessing data in a 2-dimensional array.

The **import** section of the code is slightly different than the previous programs we've studied. We have removed the **from Numeric import ***[19] line and inserted **from random import ***. The **width** and **height** statements are similar to previous programs and **axrng** is set to **2.0**. The **def init():** function specifies a white

[17] Remember that we usually start counting with "0" in computer science.
[18] We would have used a similar loop construction to store values, such as those generated by a function, prior to plotting the array.
[19] We could have left this statement in the code without problems. We didn't need the extra math functions in this example, so I chose to remove the **from Numeric import *** line.

background with black as the graphics plot color. The major changes occur in the **`def plotfunc():`**, so we'll focus on that routine.

```
def plotfunc():
     # Store the triangle vertices in an array
     verts = [[0.0,2.0],[-2.0,-2.0],[2.0,-2.0]]

     # Choose an initial point... any point
     x = -1.5
     y = 0.75

     glClear(GL_COLOR_BUFFER_BIT)
     glBegin(GL_POINTS)

     for n in range(0, 100):
          v = randint(0,2)
          x = (x + verts[v][0])/2
          y = (y + verts[v][1])/2

          if n > 30:
               glVertex2f(x,y)
     glEnd()
     glFlush()

# End Function
```

We have been discussing arrays, so it should come as no surprise that the first uncommented line of the **`def plotfunc():`** function uses an array to store the vertices of the Sierpinski Gasket. This line of code looks a bit odd compared to the x, y coordinates array example, so an explanation will be provided.

```
     verts = [[0.0,2.0],[-2.0,-2.0],[2.0,-2.0]]
```

We use the variable **`verts`** to hold the triangle vertices.[20] Look carefully at the use of brackets imbedded within brackets.[21] Although it may not be apparent at first glance, this is a 2-dimensional array. We are grouping three sets of two values each and storing these sets of values (x, y coordinates) in the **`verts`** array.

You can visualize this storage as follows:

verts array

```
     verts[0] = [0.0,   2.0]
     verts[1] = [-2.0, -2.0]
     verts[2] = [2.0,  -2.0]
```

[20] Variable names should, where possible, describe the data they store.
[21] Using an all-bracket array construction allows us to change array values during a program run. We could have used all parentheses or brackets with imbedded parentheses, but then we could not make any changes to the array while the program is running.

Compare the above table with:

```
verts = [[0.0,2.0],[-2.0,-2.0],[2.0,-2.0]]
```

verts[0] verts[1] verts[2]

Do you see how the **verts** array is structured? It is a list of ordered pairs.[22]

After storing the triangle vertices in **verts**, we then choose an initial point to begin the Sierpinski Gasket attractor. The point can be any point within reason. You could even use a couple of **random()** statements here (we'll save that for an exercise). I chose **x = -1.5** and **y = 0.75**, but again, any point would suffice. After the initial point is assigned, the graphics window is cleared and the **glBegin(GL_POINTS)** command tells OpenGL to prepare to plot pixels.

The loop statement **for n in range(0,100):** serves only to help us plot exactly 100 pixels.[23] In this loop, **n** serves only as a counter to keep track of how many times we've looped. We'll see why **n** is important in a moment. Changing the 100 to 1000, 10000, or 100000 will increase the number of plot points accordingly. Immediately following the **for** loop statement we see:

```
v = randint(0,2)
```

which chooses a random integer from 0 to 2 inclusive and stores this integer in the variable **v**.[24]

Do you remember the midpoint formula from algebra? The midpoint (x_2, y_2) of the segment joining the two points (x_0, y_0) and (x_1, y_1) is found by the following set of formulas:

$$x_2 = (x_0 + x_1)/2$$
$$y_2 = (y_0 + y_1)/2$$

Remember the midpoint formula as we explore the following two lines of code:

```
x = (x + verts[v][0])/2
y = (y + verts[v][1])/2
```

We are using the midpoint formula in the two lines of code above,[25] but it may not be readily apparent. You can see in each line that we are dividing an expression in

[22] We'll revisit these two examples again with additional information later.

[23] In this case, we don't use the index **n** to access array variables as we did in the array example using (x, y) coordinates. **n** is used only as a "counter" to insure that we actually do loop the specified number of times in order to plot the specified number of pixels.

[24] "Inclusive" means that a 0, 1, or 2 will be chosen and stored in **v** each time we "loop" by this statement. So any integers used as parameters in **random(a,b)** will randomly choose integers from **a** to **b** inclusive.

[25] Remember that we plot the Sierpinski Gasket by finding the midpoint from the current point to a random vertex, with each new midpoint serving as an endpoint for the next calculation or *iteration*.

parentheses by 2, but the parenthetical expressions look a bit strange! Let's see if we can decipher this code, starting with:

```
x = (x + verts[v][0])/2
```

But before we go further, let's make the concept of an equation in Python more clear. Anytime we see an "**=**" sign in an equation, we must interpret the expression as follows: First, calculate the value of the function on the RIGHT side of the "**=**" sign and THEN store that value in the variable on the LEFT side of the "**=**" sign. Using this rule allows us to use equations in Python (and other languages) that make no sense in algebra. For example, the equation **`x = x + 1`** has no solution. Such an equation makes perfect sense in Python, however, because we would interpret such an expression as follows: Add 1 to the current value of **`x`** and store this result as the NEW value for x. Such a construction acts as a counter because each time we "hit" the statement, 1 is added to the current value of **`x`**.[26] Again, this **`x = x + 1`** statement is an example of the concept and process of iteration, which can be generally illustrated by the following expression:[27]

(what you now have) = (what you had) + (something new)

Using this reasoning, we evaluate the statement **`x = (x + verts[v][0])/2`** by first calculating the expression on the right hand side of the "=" sign. This expression, **`(x + verts[v][0])/2`**, uses parentheses to group a couple of terms, so let's focus on **`(x + verts[v][0])`** first. To the current value of **`x`**, we add **`verts[v][0]`**. Based on the description of how to plot a Sierpinski Gasket, you might deduce that **`x`** represents the <u>current</u> **`x`** coordinate location and **`verts[v][0]`** represents the **`x`** coordinate of the randomly selected vertex… but how do we know this? Look at this earlier "visual" again with some additional information. The **`[0]`** and **`[1]`** labels are placed above the **`x`** and **`y`** coordinates respectively.

```
                [0]   [1]      [0]    [1]     [0]    [1]
verts = [[0.0,2.0],[-2.0,-2.0],[2.0,-2.0]]
                  verts[0]       verts[1]       verts[2]
```

Does this make sense? Remember that each of the vertices of the Sierpinski Gasket are stored in the **`verts`** array, shown again below (again, with additional information). Each vertex contains two pieces of information, an **`x`** and a **`y`** coordinate and we must have a method of "finding" or calling each piece of data in order to calculate a midpoint. If we label each vertex with a **`verts[0]`**, **`verts[1]`**, or **`verts[2]`**, it makes sense that we can label the **`x`** and **`y`** coordinates with an additional **`[0]`** and **`[1]`** respectively in the following manner: **`verts[0][0]`** or **`verts[2][1]`**. Do yourself a favor and examine the following table carefully!

verts array	**x**	**y**

---

[26] **`x = x + 1`** is one form of a counter. Another counter form which leads to identical results is **`x += 1`** which is sometimes called an increment function.

[27] I've already discussed the concepts of what the "=" sign does and iteration, but indulge my tendency to repeat important concepts. Iteration will be discussed again a bit later.

```
verts[0] =      [0.0,       2.0]
verts[1] =      [-2.0,     -2.0]
verts[2] =      [2.0,      -2.0]

                 [0]        [1]
```

So, **verts[0][0]** holds the **x** coordinate of the first vertex (x = 0.0) and **verts[0][1]** holds the **y** coordinate of the first vertex (y = 2.0).[28] What **verts** expression holds the **y** coordinate of the 3rd vertex and what is its value?[29] What **verts** expression contains the x coordinate of the 2nd vertex and what is its value?[30] Now we can interpret the two lines of code that calculate the midpoints.

```
x = (x + verts[v][0])/2
y = (y + verts[v][1])/2
```

In the first line (look at the right side of the "**=**" first!), we add the current **x** coordinate position to the **x** coordinate of the randomly chosen vertex stored in **verts[v][0]**.[31] We then divide this sum by 2 and store the value of the **x** coordinate of the **new** midpoint back into the variable **x**. In the second line, we add the current **y** coordinate position to the **y** coordinate of the randomly chosen vertex stored in **verts[v][1]**. We then divide this sum by 2 and store the value for the **y** coordinate of the new midpoint back into the variable **y**. So, each **new** value for **x** and **y** is calculated from the old values for **x** and **y**.[32] This procedure is an example of the iteration process we discussed earlier.

Although it isn't legal Python, we might think of these iterated equations in "pseudo-code" for clarity:

```
$x_{new}$ = ($x_{old}$ + verts[v][0])/2
$y_{new}$ = ($y_{old}$ + verts[v][1])/2
```

Now that we've calculated the new midpoint, we need to plot it using **glVertex2f(x, y)**.

```
if n > 30:
     glVertex2f(x,y)
```

The only difference between this code and previous programs is the **if n > 30:** conditional statement. We don't begin to plot points until we have passed 30

[28] THINK about this! This is a crucial concept.
[29] If you answered **verts[2][1]** and **y = -2.0** then you were correct!
[30] **verts[1][0]** and **x = -2.0**. Were you correct? If so, then you understand this array. Nice work!
[31] Remember that "**v**" will equal 0, 1, or 2 based on the **v = randint(0,2)** statement.
[32] So, **x** and **y** serve "double" duty. When on the right side of the **=** sign, they represent the old values for **x** and **y**. When on the left side of the **=** they store the new calculated values for **x** and **y**. This double duty can be confusing for humans, but Python has no problem distinguishing the "old" from the "new". See the pseudo-code equations in the text.

iterations (**`if n > 30:`**).[33] This line of code is not strictly necessary,[34] but it serves to "pretty up" the plot. The Sierpinski Gasket is an *attractor* in that eventually all points end up belonging to the set of points that make up the Gasket.[35] Since we picked the initial point at random, the first few calculated midpoints may not yet be on the Sierpinski attractor and may be found scattered in the "open" spaces of the Sierpinski Gasket. By waiting for the first 30 or so iterations to pass before we begin plotting, we will be reasonably certain that all future plotted points will be on the Gasket.

The functions **`def reshape( w, h):`**, **`def keyboard(key, x, y):`**, and **`def main():`** are essentially unchanged from previous programs with the exception of the caption in **`glutCreateWindow`** within the **`def main():`** function.

Exercises

1) Research "Sierpinski" online. See if you can find examples of other methods for creating a Sierpinski Gasket or Sierpinski Triangle. What other facts about Sierpinski Gaskets can you find? How about other fractals called Carpets? We haven't yet studied 3D graphics, but can you find an example of a 3D Sierpinski Sponge? Also, look up "Menger Sponge" and see if you notice any similarity with the Sierpinski Sponge.

2) Research "fractals" online. Be prepared to find a LOT of information on this topic! What are fractals? Why does the Sierpinski Gasket qualify as a fractal? What other fractals did you find? You might see some of the fractals you found online later in the text? As you research, remember the mathematics is NOT an opinion or belief. How might you verify the information you find online?

3) Comment the following lines in **`def plotfunc():`** as follows:

   ```
   #x = -1.5
   #y = 0.75
   ```

   and add these lines immediately below[36] **`#y = 0.75`** and above the loop code block.

   ```
   x = random()
   y = random()
   ```

   What do you think these new lines do? Run the program and see if there are any differences in the new plot when compared with the original program.

[33] Actually, 31 iterations. Why?
[34] If you remove it (and we will in an exercise) remember to move the **`glVertex2f`** indentation to match the lines above it.
[35] They are "attracted" to the Gasket… the points have no choice but to find Sierpinski!
[36] You can add a blank line first to separate the new lines from the remarked code.

4) 100 points is far too small for an adequate plot of a Sierpinski Gasket. Try 1000, 10000, and 100000 and see how the plot changes. If you have the patience, try 1000000 points! Does the graph change? If so, how?[37]

5) Remove or comment the **`if n > 30:`** line, remembering to indent the **`glVertex2f`** line at the same level as the lines above it. Run the program plotting at least 100000 points. Do you see any "stray" points? These "strays" would represent the initial point and the first few midpoints as they find their way to the Sierpinski Gasket attractor. Based on your experimentation, do we need to wait until 30 iterations have completed before plotting or would a smaller number suffice? What is the smallest value needed? Hint: You may want to plot the first few points in a different color or size (or both) to distinguish them from the points on the Sierpinski Gasket. How would you do that?

6) Place the line

```
glColor3f(random(), random(), random())
```

immediately above the **`glVertex2f`** statement. Before running the program, try to predict the output. Were you correct? What happens if you add a **`glPointSize(2.0)`** above the **`glBegin(GL_POINTS)`** statement?

7) It's now time to change the plotting rules and see if we can create something different. In **`def plotfunc():`**, change the **x** and **y** assignment statements to:

```
x = (tan(x*y*y) + verts[v][0])/2
y = (tan(y) + verts[v][1])/3
```

You should see something similar to Figure Ex. 7 at the end of these exercises.

Now try the following **x** and **y** assignment statements (note the additional code and the change in the **y** assignment statement):

```
x1 = x
x = (tan(y*y*x) + verts[v][0])/2
y = (tan(x1*x) + verts[v][1])/3
```

The **`x1 = x`** line retains the "old" value of **x** so that we can use it in the "**`y = "`** assignment statement. Otherwise, we would be forced to use the "new" value of **x** calculated in the "**`x = `**" assignment statement. This is OK if that's our intent, but sometimes we need to use the previous value of **x** or some other variable in future calculations.

Experiment with these lines and create something new!

8) Now let's change the vertices and see what we can create. Try the following:[38]

[37] You know where to change these values, don't you…? I knew you did!

```
verts = [[0.0, 2.0],[1.732, 1.0],[1.732, -1.0],
     [0.0, -2.0],[-1.732, -1.0],[-1.732, 1.0]]
```

Let **axrng = 1.0** and in **def plotfunc():** change the **randint()** statement to:

```
v = randint(0,5)
```

Also, change the **x** and **y** assignment statements to:

```
x = (x + verts[v][0])/3.0
y = (y + verts[v][1])/3.0
```

Note that you are dividing by **3.0** instead of **2.0**. Can you predict what will happen when you run the program? You should see something like Figure Ex. 8 at the end of these exercises. Surprised?

9) Once again we'll change the x and y assignment statements. We'll use the same lines of code that we used at the beginning of exercise 7 as follows:

```
x = (tan(y*y*x) + verts[v][0])/2
y = (tan(y) + verts[v][1])/3
```

Let **axrng = 1.5** and run the program. The output should look like Figure Ex 9. The plot looks a bit "plant-like", don't you think? Remember this structure in the next section of the text!

10) Finally, try one more modification and then you can experiment on your own. Again, set **axrng = 1.5** and make a simple modification to the "**x =** " assignment statement as follows:

```
x = (tan(y*x) + verts[v][0])/2
```

The plot is interesting and should resemble Figure Ex 10. Now experiment on your own and see what you can create.[39]

11) Ok, I didn't really mean "Finally" in the last exercise. I simply had to add one more, mainly because I think this one is really neat. I'm not certain what to call it, so how about "Blank's Carpet"? Seriously, try the following modifications to your program:

```
# Initial values
width = 500
height = 500
```

[38] This is all one statement and it's OK to type it as such… but it's also OK to type it as seen in the text. Python doesn't care and will ignore the line feed. You could even carry this example further and put a single vertex on each line for clarity. Try it!
[39] Iterated functions that create pictures such as the ones in this section and the Barnsley Fern in the next section are called Iterated Function Systems.

```
axrng = 1.0

# in plotfunc
verts = [[-2.0, 2.0], [-2.0, -2.0], [2.0, 2.0],
     [2.0, -2.0],[-1.0,1.0],[-1.0,-1.0],
     [1.0,1.0],[1.0, -1.0]]

for n in range(0,100000):
     v = randint(0,7)
     x = (x + verts[v][0])/3.0
     y = (y + verts[v][1])/3.0
```

Place the above lines in the proper locations in your program code. Notice that we have added some extra vertices. Where would they be plotted? Would all of them show up in your graphics display window? Before you run the program, do you have any idea what the fractal will look like? Notice that we are choosing points 1/3 the distance from the current location to the next random vertex. Go ahead and run the program! You should see something like Figure Ex 11 on the next page. I think this is an interesting plot (at least).

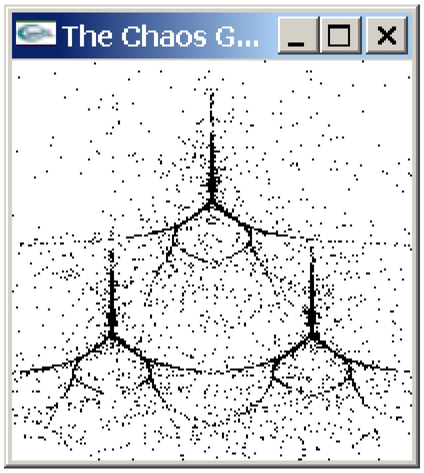
12) Can I add one more exercise? You'll like this one, I promise! Modify the code from the previous exercise as follows:

```
# Add the following import statement
# to get the trig functions
from Numeric import *

# Initial values
axrng = 10.0

# In plotfunc
verts = [[-2.0, 2.0], [-2.0, -2.0], [2.0, 2.0],
          [2.0, -2.0]]
```

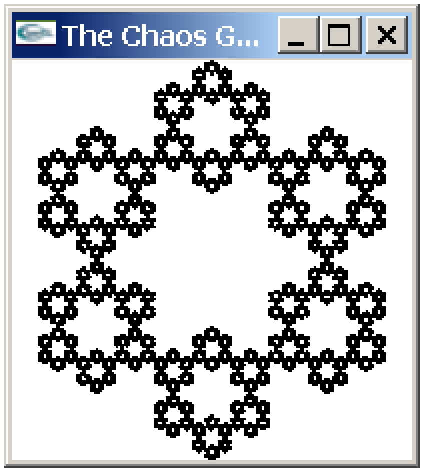
and change the **for** loop to:

```
for n in range(0,100000):
     v = randint(0,3)
     x = (x + verts[v][0])*sin(y)
     y = (y + verts[v][1])*cos(x)
```

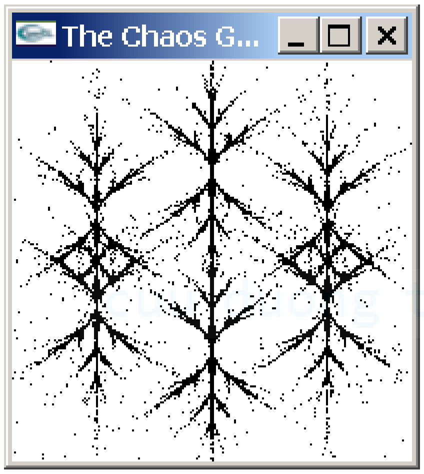
The multiplication of the vertex selection code by trig functions should add some interesting effects to the graphic display. Run the program and see what happens. I think this is a beautiful plot. Can you add some color and make it even better? Figure Ex 12 shows the fractal!



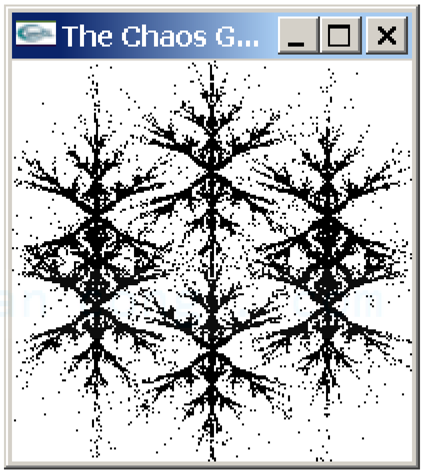
Ex 7



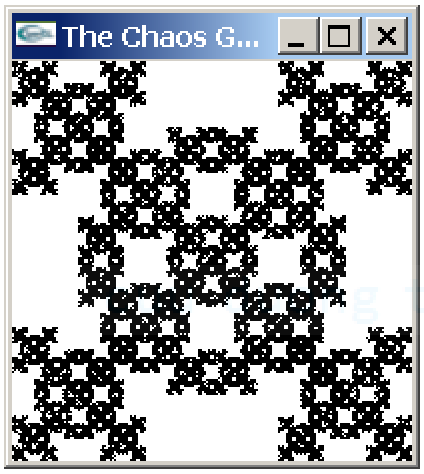
Ex 8



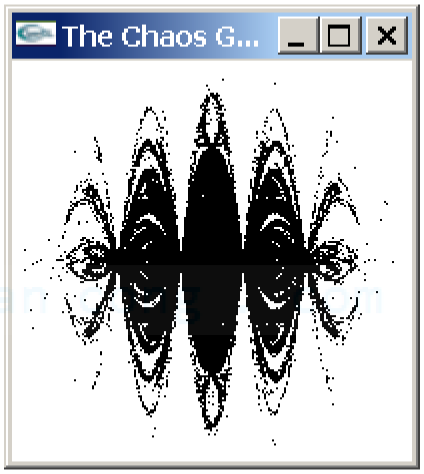
Ex 9



Ex 10



Ex 11



Ex 12

Section 6.4 The Barnsley Fern[40]

Exercise 9 in the previous section asked you to remember the plant-like structures that were plotted when you added and modified the lines of code listed in the problem. We are now going to expand on the ideas introduced in the Chaos Game section to produce an graphics plot that raises the “plant-like” descriptor to a new level. In Section 6.3 we used a simple rule with a random choice of vertices to produce the Sierpinski Gasket attractor. In this section we are going to enlarge our scope and use a combination of 4 sets of parameters, each one assigned a different probability of being selected at random. Once a parameter set has been randomly chosen we’ll apply the selected parameters to our rules or equations. The result of this iterated function system is both surprising and beautiful!

Here is the program listing for this section.

```
# PyBarnsleyFern.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from random import *
from Numeric import *
import sys

# Globals for window width and height
global width
global height

#  Initial values of width and height
width = 600
height = 600

def init():
      # White background
      glClearColor(1.0, 1.0, 1.0, 0.0)

      # Green Plot… it IS a Fern
      glColor3f(0.3, 0.6, 0.2)

      #  Set the projection matrix... our "view"
      glMatrixMode(GL_PROJECTION)
      glLoadIdentity()

      #  Set the plot window range
      gluOrtho2D(-3.0, 3.0, 0.0, 10.5)

      #  Set the matrix for the object we are drawing
      glMatrixMode(GL_MODELVIEW)
      glLoadIdentity()
```

[40] See http://mathworld.wolfram.com/BarnsleysFern.html

```
def plotfunc():

     # Choose an initial point... any point
     # You can randomize this if you wish
     x = -1.5
     y = 0.75

     glClear(GL_COLOR_BUFFER_BIT)
     glBegin(GL_POINTS)

     # Plot 100000 points.  This number is very large.
     # Feel free to experiment with smaller values.

     for n in range(0,100000):

          # n allows us to reject the first few points
          # to give the attractor a chance to do its “thing”

          # Choose a random value between 0 and 1 and
          # then select a set of parameters based on this value.
          v = random()

          if v >= 0 and v <= 0.8000:
               a = 0
               b = 1.6
               c = -2.5*pi/180
               d = -2.5*pi/180
               e = 0.85
               f = 0.85
               #glColor3f(1.0, 0.0, 0.0)

          elif v > 0.8000 and v <= 0.8050:
               a = 0
               b = 0
               c = 0*pi/180
               d = 0*pi/180
               e = 0
               f = 0.16
               #glColor3f(0.0, 1.0, 0.0)

          elif v > 0.8050 and v <= 0.9025:
               a = 0
               b = 1.6
               c = 49*pi/180
               d = 49*pi/180
               e = 0.3
               f = 0.34
               #glColor3f(0.0, 0.0, 1.0)

          elif v > 0.9025 and v <= 1.0:
               a = 0
```

```
                b = 0.44
                c = 120*pi/180
                d = -50*pi/180
                e = 0.3
                f = 0.37
                #glColor3f(1.0, 0.0, 1.0)

           #  Save the old values of x and y so we can iterate
           #  those values according to the chosen parameters
           #  and rules.
           xx = x
           yy = y

           #  Apply the parameters to the rule equations
           x = e * xx * cos(c) - f * yy * sin(d) + a
           y = e * xx * sin(c) + f * yy * cos(d) + b

           #  Start plotting after the 10th point
           if n > 10:
                glVertex2f(x,y)
      glEnd()
      glFlush()

def keyboard(key, x, y):
      #  Allows us to quit by pressing 'Esc' or 'q'
      if key == chr(27):
           sys.exit()
      if key == "q":
           sys.exit()

def main():
      global width
      global height

      glutInit(sys.argv)
      glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
      glutInitWindowPosition(100,100)
      glutInitWindowSize(width,height)
      glutCreateWindow("The Chaos Game... Fern!")
      glutDisplayFunc(plotfunc)
      glutKeyboardFunc(keyboard)

      init()
      glutMainLoop()

main()

#End of Program
```

Figure 6.3 at the end of the exercises illustrates the Barnsley Fern. Nice, huh? This is supposedly an example of a spleenwort fern, but I'm not a fernologist[41] so I can't verify this. The Barnsley Fern is another example of a fractal image. Remember that fractals are self-similar geometric objects that have fractional dimensions.[42] Self-similar means that as we magnify the fractal, the details never diminish and we continue to see the same patterns, perhaps with slightly different rotations or with some distortions. In the case of the Barnsley Fern, each branch is a smaller copy of the entire fern. Each leaf on a branch is a smaller copy of the branch, which is a smaller copy of the entire fern… and so it goes! Fractals are infinite in complexity, yet they can be captured to an extent by your computer. The next chapter will be devoted to some of the more beautiful easily generated fractal images. Toward the latter part of the text, we'll make a valiant attempt to visualize some 3D fractal images and perhaps some 4D fractals. Some of these are bizarre in their structure and I hope you'll enjoy them!

The Barnsley Fern program in this section randomly chooses a set of parameters and then applies those parameters to some specific rule statements. With the exception of the **`init`** function, which sets the **`x`** and **`y`** axis ranges to different values than in previous programs (look at the ranges!), the majority of the Barnsley Fern program listing is similar to the Sierpinski Gasket "Chaos Game" program. We'll focus our attention only on the section of code in the **`def plotfunc():`** routine where the fern attractor is created. The **`v = random()`** statement assigns a pseudo-random value between 0.0 and 1.0 to the variable **`v`**. Notice the series of **`if… elif`**[43] decisions following the **`v = random()`** line of code. Careful examination should reveal that the first **`if`** statement and its associated parameter set is far more likely to be chosen than any of the other conditional statements. Why? Which conditional or decision statement and parameter set is least likely to be chosen? Again, why? Notice the use of **`pi`** within the conditional statement code blocks. We are using **`pi`** in the parameter sets to convert from angle measurements (which the Python trig functions can't use) to radian measurements (which the Python trig functions MUST use). There are **`pi`** radians in 180 degrees, so the conversion is:

n degrees **`*`** (**`pi`** radians)/(180 degrees)

The degree(s) units will cancel and we are left with the appropriate angle measure in radians. So, **`c = 120*pi/180`** is a statement that converts 120 degrees to radians and stores the radian angle measure in the variable **`c`**. Which other parameter uses this conversion?[44]

The rule statements:

```
x = e * xx * cos(c) - f * yy * sin(d) + a
y = e * xx * sin(c) + f * yy * cos(d) + b
```

[41] I'm certain that isn't the correct term for a fern expert.
[42] Instead of 2D, the dimension may be 1.576D. We'll touch on the topic of fractals, but a detailed study of these interesting and beautiful objects is beyond our scope at this point.
[43] **`elif`** is a Python abbreviation for "else if".
[44] "**d**" of course!

both use the randomly chosen parameters and the previous values of **`x`** and **`y`** (**`xx`** and **`yy`**) to calculate new values for **`x`** and **`y`**. These new values are then plotted using the **`glVertex2f`** command. Notice the use of trig functions within the equations. These trig functions make use of the radian angles converted using **`pi`** in the parameter code blocks. You also probably noticed the commented **`glColor3f`** statements after each parameter set in the **`if… elif`** decision blocks. We'll explore the affect of these **`glColor3f`** statements (as well as other options) in the exercises. Make certain that you save this program prior to working with the exercises. You can then reuse the saved code for each exercise.

Exercises

1) Uncomment each of the **`glColor3f`** statements in the **`if… elif`** decision block. You can now identify which parameter set is responsible for each part of the fern attractor. Are you surprised? Feel free to experiment with different colors schemes.

2) Change the number of iterations from 100000 to a much smaller number (say, 1000) and see what effect this has on the fern attractor. What is the minimum number of iterations needed to create a decent (to you) attractor? Likewise change the number of iterations to a larger value. The simple fact that all points eventually find their way to the image of the fern provides a nice intuitive definition of the term "attractor".

3) Above the **`glBegin(GL_POINTS)`** statement, add **`glPointSize(2.0)`** and see what happens. Try changing **`glBegin(GL_POINTS)`** to **`glBegin(GL_LINES)`** or **`glBegin(GL_LINE_STRIP)`** and see what effect this has. Do you like the result?

4) Experiment with some of parameter assignments. See if you can make the fern bend or twist. Perhaps you can even change the leaf or stem patterns?

5) Change the probabilities within the **`if… elif`** decision statements. What happens? Notice that the sum of all the probability intervals equals 1.0. Why? What happens if there is a "hole" in the interval? For example, change the first **`if v >= 0 and v <= 0.8000:`** statement to **`if v >= 0 and v <= 0.5000:`** and see what happens. Was there a difference in the fern plot? What did you expect to see?

6) One simple modification that you can make is to change the **`sin()`** statements in the rules to **`tan()`** statements. At small angles, the values of **`sin()`** and **`tan()`** are nearly the same, so distortions would occur only when the angles are relatively large. The figure "**tan()**" at the end of the exercises illustrates this simple change. The effect is not dramatic, but the fern appears to be a bit more filled out in its foliage.

7) Experiment with the rule statements and see if you can create a unique and perhaps otherworldly plant. Make certain you save your best efforts!

8) The Barnsley Fern program does not have a **`reshape`** function. Is this a problem? What difficulties can you foresee in adding a **`reshape`** function to the program? Would it be impossible to add such a function?

9) Use Google or your favorite search engine to look for “Michael Barnsley” and “fractals” and see what you can find. Barnsley is one of the pioneers in fractal imagery. In particular, see if you can find references to his Collage theorem. What is this theorem used for? You might remember this reference for a final project suggestion if you are interested in the topic of fractal forgeries.

10) This is just a thought question. How does nature determine patterns in living organisms? The easy answer is "By using the genetic code in DNA". But how does this code work? Could it be that there is an iterated system found in the DNA of every living thing? All maple leaves are different, yet they all are similar in shape and structure otherwise we could not distinguish a maple leaf from an elm leaf. Can this be explained by assuming that DNA randomly chooses a set of parameters (whatever they may be) and applies a set of rules to those parameters? Look up Stephen Wolfram and "A New Kind of Science" to see what one scientist has to say about nature and computation.

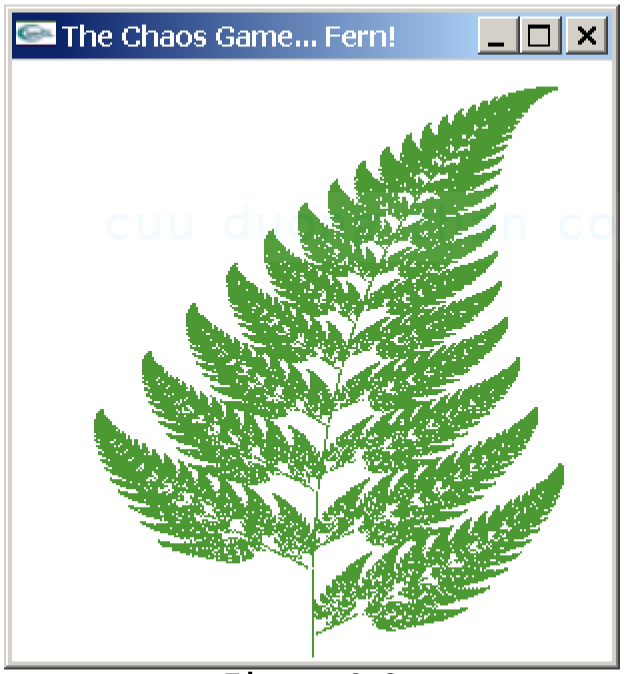
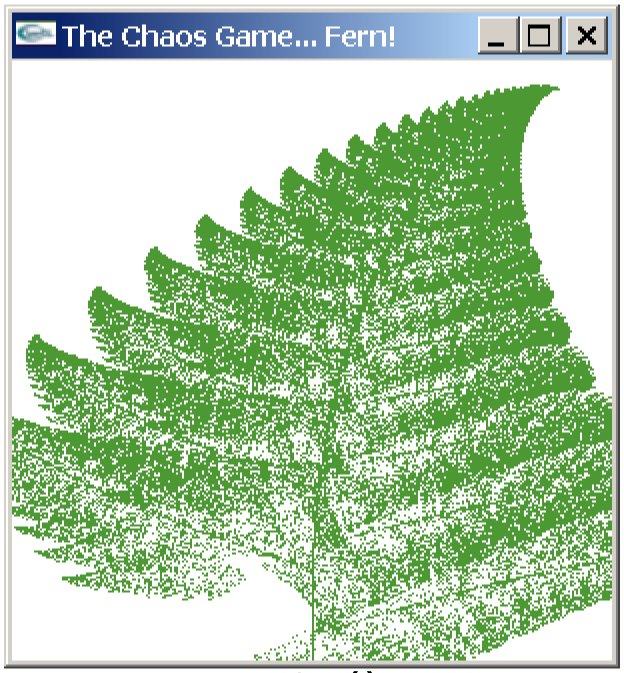


Figure 6.3



tan()

Section 6.5 Chaos and the Logistic Map

The population of a living species may fluctuate from year to year depending on the available resources and the environmental conditions of the habitat. Biologists are interested in the dynamics[45] of population growth and decline. Computer simulations have provided some insight into the nature of population dynamics and such simulations continue to be an area of research and interest to modern biologists. In this section we will conduct a classic simulation of population dynamics. We will use this simulation to introduce the topic of chaotic behavior produced from a simple deterministic equation.[46]

The logistic equation, $x_{n+1} = rx_n(1-x_n)$ has been used to model population growth and decline in addition to providing an introduction to the subject of chaos and chaotic dynamics.[47] [48] In this equation, x represents the decimal percentage of the species population in terms of the total carrying capacity of the habitat. For example, **`x = 0.85`** would mean that the population is at 85% of its maximum capacity. The parameter r represents all factors that affect the population we are studying (food, competition, climate, etc.). The subscripts n and $n+1$ indicate that the output, x_{n+1}, will be used as the input, x_n, during the next iteration. If the value of x_{n+1} is high, the term $(1-x_n)$ will be relatively low for the next calculation, resulting in a lower population in the next generation. As an example, using **`x = 0.85`** would result in **`(1- 0.85)`** or **`0.15`** as the population value for the next generation. This would represent a massive die-off within the species in that particular habitat. Likewise, using **`x = 0.23`** as the initial population would result in **`(1- 0.23)`** or **`0.77`** as the population in the next generation, presumably due to the abundance of resources available to the lower **`0.23`** number of individuals. So, the $(1-x_n)$ term acts as a feedback mechanism, insuring that under many conditions, the species population will adjust itself according to the conditions specified by the variable **`r`**.

This relatively simple equation, when iterated, produces surprising and unpredictable results. Suppose we start with a value of **`r = 2.5`**[49] and a value of **`x = 0.5`**. Iterating the logistic equation we quickly find that the "population" stabilizes at **`0.6`** of its maximum carrying capacity. We can see this by plotting the graph of the population with respect to time. In this case, time will be represented along the **`x`**-axis by

---

[45] Dynamics can be defined here as the process of change in a system. This process of change can be a change in position, motion, or simply a change in value or number. System dynamics is quantitative or numerical in nature, but can be studied qualitatively by using graphics such as we are doing here.

[46] While this isn't a precise definition, we can restate this as "apparently random behavior from a relatively simple non-random equation". The study of chaos and chaotic systems had its zenith in the late '80's and early '90's. I still find the topic fascinating and I hope you find it interesting as well.

[47] Gleick, James (1987). "Chaos: Making a New Science". Penguin Books.

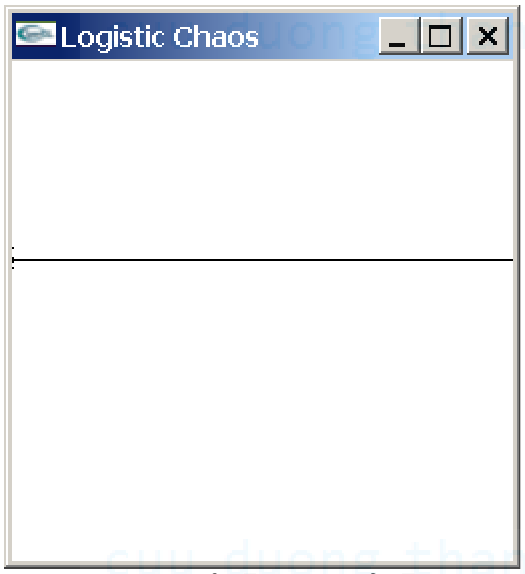
[48] Stewart, Ian (1992). "Does God Play Dice? The Mathematics of Chaos". Blackwell Publishing, p. 136.

[49] The meaning of **`r`** is a bit obscure. In the logistic equation or mapping, **`r`** simply represents all the factors that affect a species population. **`r`** is meaningless in terms of "What does **`r = 2.5`** signify?" It's just a value that represents a measureless quantity. It turns out, though, that the parameter **`r`** is responsible for the interesting behavior behind the logistic equation and the logistic equation seems to have some validity as a model of species population over time.

subsequent generations. The following program listing will plot this graph under the specified conditions.

```
# PyLogistic.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

#  Define the width and height variables as global
global width
global height

#  Initial values for width and height
width = 600
height = 600

def init():
     glClearColor(1.0, 1.0, 1.0, 1.0)
     gluOrtho2D(-8.0,7.0,0.0,1.0)

def plotlogistic():
     glClear(GL_COLOR_BUFFER_BIT)

     glPointSize(1.0)
     glBegin(GL_POINTS)

     # Set the initial values of x and r
     x = 0.5
     r = 2.5

     #  Range over the entire interval
     for a in arange(-8.0, 7.0, 0.0001):

          # The logistic equation
          x = x*r*(1-x)
          glColor3f(0.0, 0.0, 0.0)
          glVertex2f(a,x)
     glEnd()
     glFlush()

     #  Print the final value
     print x

def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'
     if key == chr(27):
          sys.exit()
     if key == "q":
          sys.exit()

def main():
     global width
     global height

     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
     glutInitWindowPosition(200,200)
     glutInitWindowSize(width,height)
     glutCreateWindow("Logistic Chaos")
     glutDisplayFunc(plotlogistic)
     glutKeyboardFunc(keyboard)

     init()

     glutMainLoop()

main()

# End Program
```

The plot from this program is found in the figure below.



x = 0.5 r = 2.5

Not very impressive, is it? There appears to be a bit of “noise” at the extreme left hand portion of the screen. This noise represents some initial population fluctuations resulting from the **(1-x$_n$)** feedback mechanism attempting to find equilibrium. Rather quickly 0.6 is reached as a fixed point. No further change in the population value occurs once this fixed point is established. You should see 0.6 displayed in the DrPython console.

The program listing is fairly self-explanatory. The logistic equation is found in the **`plotlogistic`** function and is properly commented. Study this function carefully so that you understand how the logistic equation is iterated. As in previous iterated functions we've encountered, the "new" calculated value of the function (stored in **`x`**) will be used as input for the next iteration (it will then be the "old" value at that future time). The logistic equation is a dynamic system, meaning that it is constantly changing based on the current conditions. Nature enacts such changes continuously, but we can't model continuous time flow using a computer. Instead, we introduce increments of time (iterations) to model nature as closely as possible. The smaller the increments (usually resulting in more iterations), the better we may simulate nature. Unfortunately, when we decrease the time increments and increase the number of iterations, our simulation can slow significantly. In this program we are using **`0.0001`** as our increment length. You can experiment with the size of this value to see the effect on the simulation speed. If the increment is too large, the plot will appear "dotted". If the increment is too small, we may unnecessarily increase the execution time of the program. It is possible to calculate the optimum increment in most cases.[50] We'll discuss this topic further in later chapters.

Note the **`print x`** statement following the **`glFlush()`** command. This statement is used to print the value corresponding to the population in this first example. You might loosely define this value as the current decimal representation of the fraction of the carrying capacity of the habitat based on the present conditions (initial population and the value for **`r`**). If the population fluctuates, this statement will only print the last calculated value in the simulation. If you desire a running total, then you can place this statement immediately after the logistic equation **`x = x*r*(1-x)`**. Be warned that this will slow down program execution! My recommendation is that you comment out this line after the first simulation.

Here is a short deviation from the task at hand. One of my former students "accidentally" commented the **`glutInitWindowSize(width, height)`** statement and noted that the program still ran. Try this! The program will run, but the window size will be the default value of 300 x 300 rather than the 600 x 600 specified in the initial **`width`** and **`height`** variables.

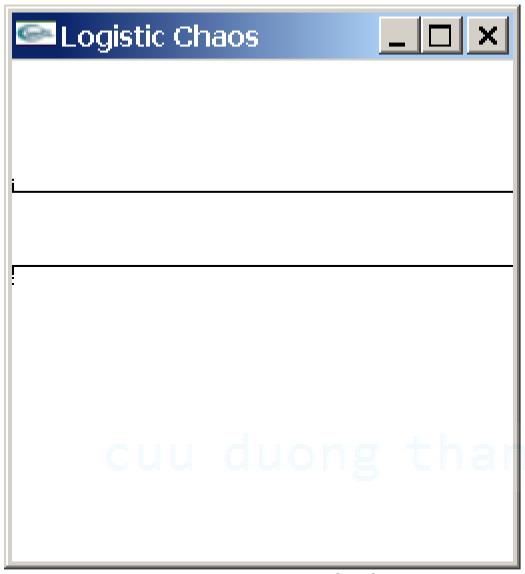
Let's experiment a bit. First change the value of **`x`** in the **`plotlogistic`** function to **`x = 0.95`** and run the program again. Try a few more initial values for **`x`** (remember that **`0.0 < x < 1.0`**, in other words, keep **`x`** between 0 and 1). It appears as if the value of **`x`** is not crucial to the outcome of the simulation! From our simulation, we can state that the initial size of the population doesn't appear to affect the long-term stability of the population using this particular value for **`r`**. Now let's change **`r`** to **`r = 3.0`** and see what happens. There is definitely a change in the graph. The line seems to be a bit thicker and the left hand side of the graph has a split appearance. The population does seem to again stabilize, but this time at a slightly higher value than before. Apparently there is some minor periodic fluctuations at the beginning of the

[50] There are instances, such as in this example, where we use a much smaller increment than needed in order to slow down the rendering of the graph. Modern computers are so fast that many drawings are plotted almost instantly. This is great if all we are interested in is the final graph. Sometimes, though, it's nice to watch the graph being plotted. In those cases, we increase the number of increments (or decrease the time interval) to slow things down.

simulation under these conditions. Once again change the value of **x** to verify that this variable has no role in the long-term behavior of the population.

To make things a bit more interesting, set **r = 3.05**. The result is pictured below in figure **x = 0.5 r = 3.05**. We implemented a small change in **r** and the result is a population that appears to fluctuate back and forth between two values! Can you find the parameter between **r = 3.0** and **r = 3.05** where the two separate population values first appear? The next figure **x = 0.5 r = 3.45** was created using **r = 3.45**. The population fluctuations are now a bit more complex. Try **r = 3.49**. Do you see that our species population now alternates between 4 separate values? Now try **r = 3.562**. The result is in figure x = 0.5 r = 3.562. Now there are 8 separate values! This is getting interesting! **Try r = 3.565** and then **r = 3.571**. What do you see? I find it amazing that such small changes in **r** result in impressive changes in population fluctuations.

One of the hallmarks of chaotic behavior is the sensitivity of such systems to the initial conditions of the simulation. In English, this means that a very small change in the initial conditions (such as the value of **r**) can result in enormous changes in the outcome. This is due to the feedback mechanism built into iterative systems. Small errors keep growing (like feedback in a microphone and speakers!) until they completely overwhelm the system. Also, notice the pattern of changes in the population values: 1, 2, 4, 8. If we were to experiment carefully, we would find 16, 32, 64, etc. This is called "period doubling" and is found in all chaotic systems, not just this one. If we were to check closely, we would also find that the rate of period doubling is constant at about 4.669[51] times quicker for each successive doubling.

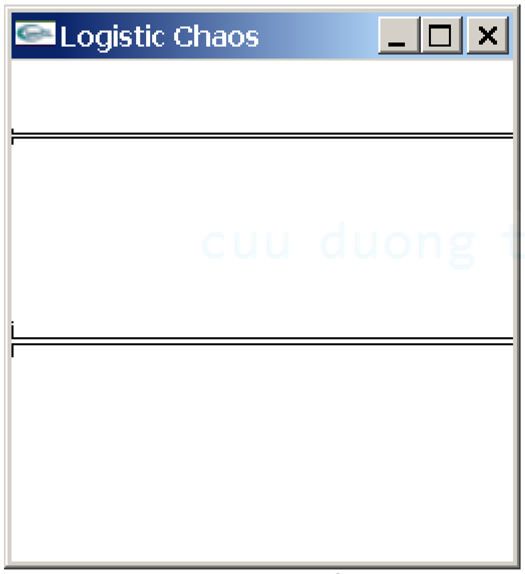


x = .5 r = 3.05

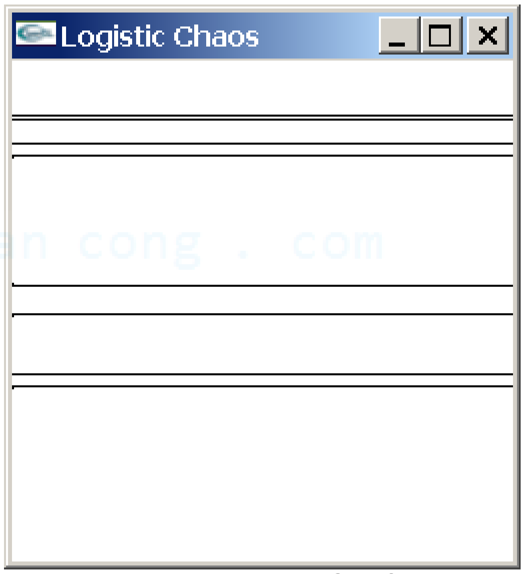
[51] This is called Feigenbaum's Constant after Mitchell Feigenbaum, who first discovered the value experimentally using a hand held programmable calculator in 1975. He found period doubling in the logistic equation increased at the rate of about 4.669 for each successive doubling. He then discovered the same 4.669 value in other completely different systems and he theorized that this value was a universal constant found in all chaotic systems. He was proven correct and the constant now bears his name.

Finally, let's glimpse true chaos. Set **`r = 3.98`**. The result is nothing short of impressive. We now have a population which is unstable at best. Even though we are using the same logistic equation as before, it would be virtually impossible to predict the population for more than a few iterations in advance. Figure **`x = 0.5  r = 3.98`** illustrates the chaos lurking within this simple equation.

To generate the Chaos! figure pictured below requires both an explanation and a modification to our program. The explanation involves varying the value of **`r`** as we move from left to right in the window. Lower values of **`r`** (<3.00) indicate a single stable population. As we increase the value of **`r`**, the stable population rises until we reach a point where the population abruptly divides into 2 values (at ~3.05). As **`r`** continues to increase, we see period doubling bifurcations[52] to 4, and then 8 values. After that, the bifurcations come too quickly to distinguish in this figure and chaos is rapidly established. This is a classic figure used frequently to illustrate explanations of chaotic phenomena. Now you know how to make this graph yourself!

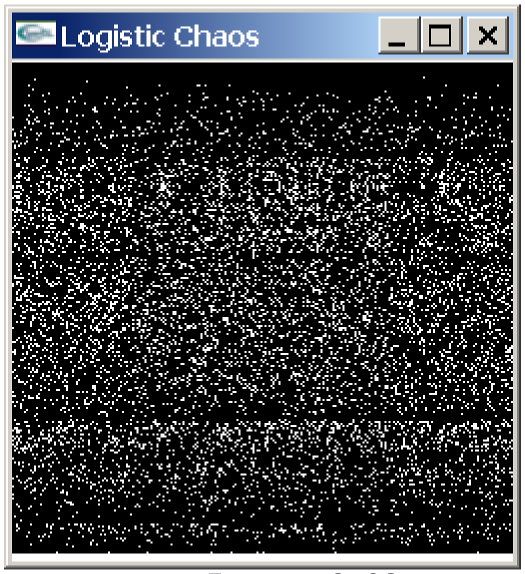


`x = .5  r = 3.45`

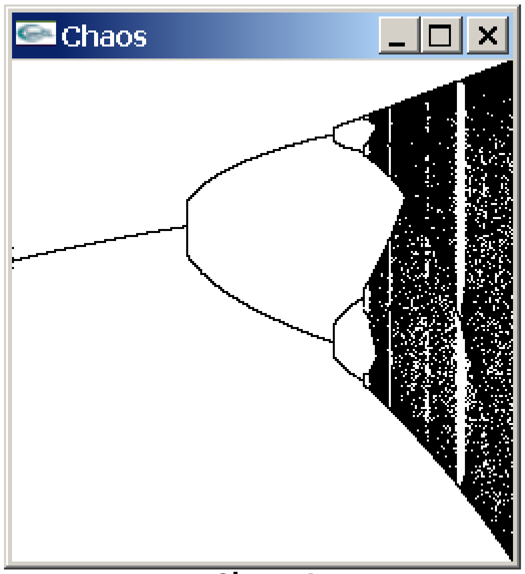


`x = .5  r = 3.561`

[52] A bifurcation is a splitting into two parts or paths.



`x = .5   r = 3.98`



Chaos!

The program listing modifications needed to render the Chaos! Figure above are found in the **`def plotLogistic():`** function as follows:

```
def plotLogistic():
     glClear(GL_COLOR_BUFFER_BIT)

     glPointSize(1.0)
     glBegin(GL_POINTS)

     x = 0.5
     r = 2.5

     for a in arange(-8.0, 7.0, 0.0001):
          r = r + 0.00001

          x = x*r*(1-x)
          glColor3f(0.0, 0.0, 0.0)
          glVertex2f(a,x)
     glEnd()
     glFlush()

# End function
```

We set **`r = 2.5`** initially and allow it to increase by 0.00001 during each iteration of the **`r = r + 0.00001`**[53] statement within the **`for`** loop. As you recall, the value of **`r`** is the critical value in our program. As **`r`** increases, the population changes until chaos is reached. You may be tempted to think that such a simple population model would have little resemblance to real populations in the wild. In fact, many species of plants,

[53] This statement can also be written **`r += 0.00001`**.

insects, and animals seem to exhibit both stable and chaotic fluctuations in population numbers as a result of changing conditions within their habitats.

Exercises

1) Use Google and look up Mitchell Feigenbaum and chaos. Also look up Robert May and chaos.

2) Another method of looking at the logistic equation is to consider that the equation is actually a quadratic function and can be graphed as a parabola that opens downward. We can see this by multiplying **`rx(1-x)`** and getting **`rx – rx`**^{2}. The negative **`rx`**^{2} term dictates the downward opening parabola. We can then iterate the equation on the parabola and **`y = x`** line and see the chaos emerge. Try the following changes to see this different view. First modify the **`gluOrtho2D`** statement in **`def init():`** as follows:

```
gluOrtho2D(0.0,1.0,0.0,1.0)
```

Once you have adjusted the viewing dimensions, then change the **`def plotlogistic():`** function according to the following listing.

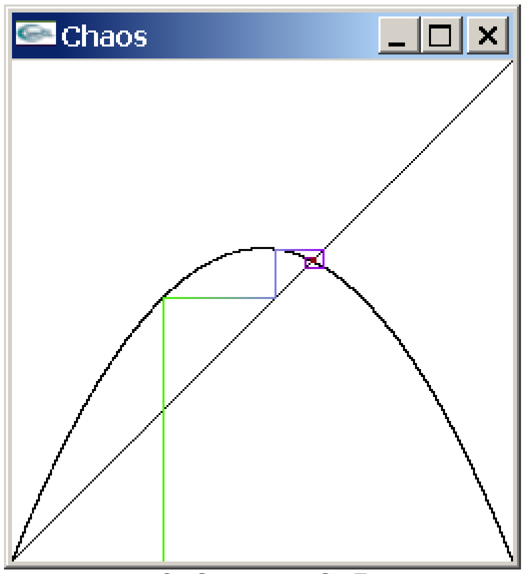
```
def plotlogistic():
    glClear(GL_COLOR_BUFFER_BIT)

    glPointSize(1.0)
    glBegin(GL_POINTS)

    # set the r parameter
    r = 2.5

    # Plot the logistic parabola and the line y = x
    for x in arange(-1.0, 1.0, 0.00001):
          y = x*r*(1-x)

          glColor3f(0.0, 0.0, 0.0)
          glVertex2f(x,y)
          glVertex2f(x,x)

    glEnd()

    # Plot the population
    x = 0.3
    y = 0.0
    glBegin(GL_LINE_STRIP)
    for n in arange(1,100000):

          # add some color for effect
          glColor3f(sin(x),cos(x*n),sin(n*y))
```

```
        # the OLD point
        glVertex2f(x,y)

        y = x*r*(1-x)

        # the NEW point
        glVertex2f(x,y)
        x = y
    glEnd()
    glFlush()

     # End Function
```

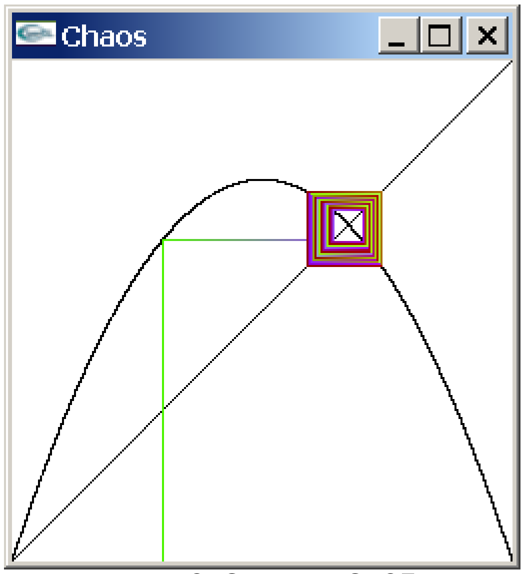
The graph is illustrated in figure **`x = 0.3 r = 2.50`** on the next page. The "upside down" parabola represents the logistic equation. The diagonal line is the **`y = x`** reference line. This reference line is important because it will allow us to use the output of the logistic equation as input for the next iteration. The interpretation of the horizontal and vertical lines connecting the parabola and the **`y = x`** reference line is a bit tricky, but remembering the concept of iteration, let's give it a try. We start with **`x = 0.3`**, represented by the vertical line extending from the bottom of the window (at a location corresponding to approximately **`x = 0.3`**) and upward until it intersects the parabola. The value of the logistic equation at this point (the intersection of the **`x = 0.3`** vertical line and the parabola) represents the new population in the next iteration. We iterate this new population back into the logistic equation by drawing a horizontal line from this intersection point to the **`y = x`** reference line and then upward until we intersect the parabola again. This process is continued until we stabilize at a single point (as shown in the first figure below) or we complete some specified number of iterations. The **`y = x`** reference line is how we turn "old" values of the logistic equation into new input for the next iteration. Do you see how this works?

If you are wondering about the duplication of **`glVertex2f(x,y)`** in the **`for`** loop, remember that we are using **`glBegin(GL_LINE_STRIP)`** to plot from the previous point to the current point. The first **`glVertex2f(x,y)`** is the previous point and the second **`glVertex2f(x,y)`** represents the current point after the logistic equation is applied. The result is a connected set of line segments that represent the path or orbit of our iterations. Graphical analysis, while not definitive in terms of proof, is a very powerful tool for analyzing the qualitative behavior of equations and systems of equations. We can actually see the logistic equation search for a stable solution (if one exists) or descend (ascend?) into chaos.

Let's experiment. Set **`r = 3.05`**. Figure **`x = 0.3  r = 3.05`** shows the result. The graph never settles to a single fixed point. If we watch closely, we may be able to see that the horizontal and vertical lines eventually intersect the parabola in two different points, corresponding to two population values. This two cycle behavior is sometimes called a limit cycle. A limit cycle is simply a cycling or repeating set of values and can be of any length depending on the equation and equation parameters.

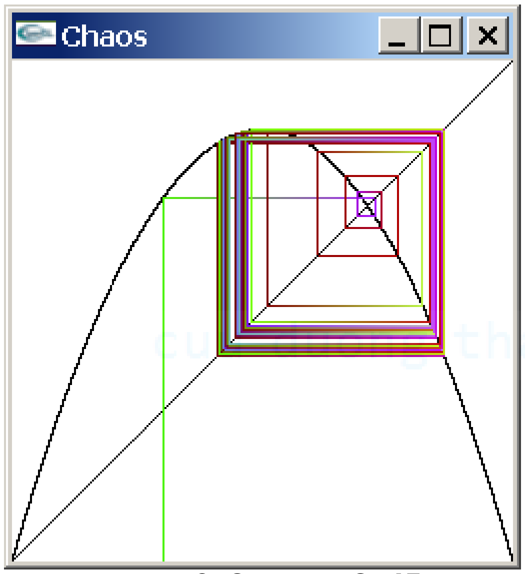


x = 0.3 r = 2.5

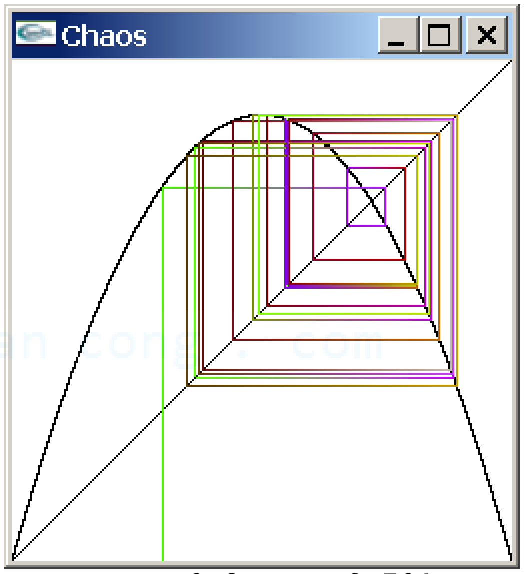


x = 0.3 r = 3.05

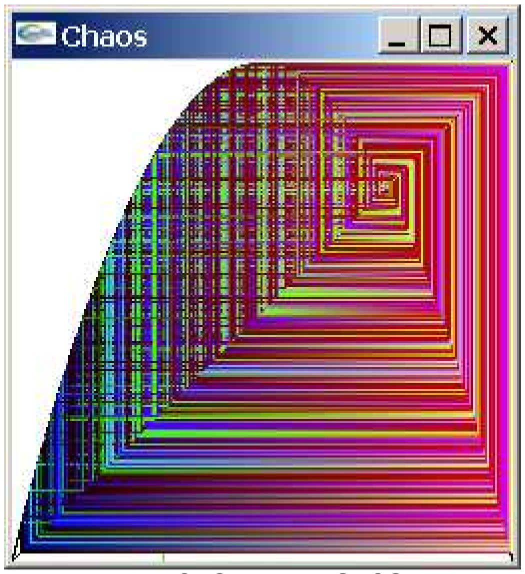
Now let's increase **r** until we approach chaos. Try **r = 3.45**. The result is shown in figure **x = 0.3 r = 3.45** below. Notice that the graph fluctuates around several values. **r = 3.561** demonstrates even more fluctuations as pictured in **x = 0.3 r = 3.561**. Finally, to get a picture of full blown chaos, set **r = 3.98** and run the program. The result is rendered in figure **x = 0.3 r = 3.98** and illustrates the wild population values created as a result of the chaotic behavior of the equation. The figure can't do justice to actually watching the program run. You can easily see the population size change in a seemingly random fashion and cover <u>almost</u> the entire range of allowable values. What values appear to be left out?



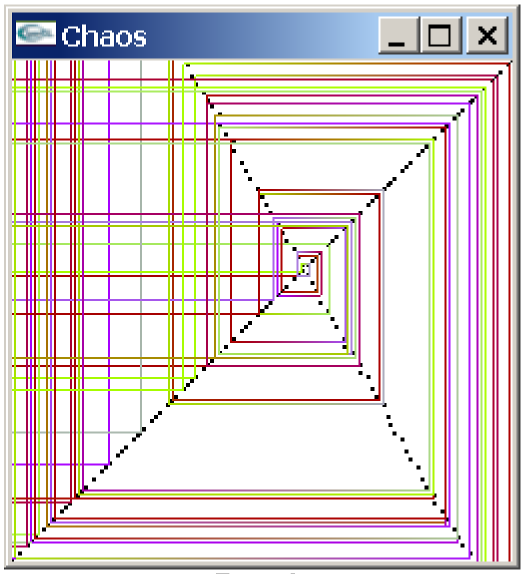
x = 0.3 r = 3.45



x = 0.3 r = 3.561



x = 0.3 r = 3.98



Zoom!

3) Try several values of **r** between 3.00 and 4.00. Try **r = 4.00** and see if you can detect a difference between **r = 4.00** and **r = 3.98**. What is this difference? Finally, try **r = 4.01** and see what happens. Can you explain the result?

The figure Zoom! Illustrates a magnification of the intersection point between the **y = x** line and the parabola in the previous figure. This magnification is around 100000 times, certainly in the realm of an electron microscope! Notice that the orbits of the chaotic lines encircle the intersection, but of the 5 million iterations plotted for this diagram (most of which are much larger than this very tiny area and are not shown), none appear to hit the intersection point. This is certainly not a proof, but it is suggestive that no line will ever strike this intersection point even though the iterations are wildly chaotic. Can you think of a reason why this is so?[54] Along the same "lines" (sorry), none of the orbits will ever retrace itself. Why not?[55] So, even though the logistic system exhibits chaos and has the appearance of randomness, there are some constraints on the system. We'll visit this topic again in the next chapter. Can you figure out a way to modify your program to replicate the **Zoom!** figure?

This is not an exercise, per se, but I want you to think about this! Computers have a reputation for truth. The computer is never wrong.[56] If you don't believe me, then the next time you are incorrectly billed or scheduled for a class you didn't sign up for… or given a grade you didn't deserve, then where will the blame for the error be placed? Most likely the error will be blamed on the human who input the original

[54] What would happen if one of the chaotic orbits would exactly hit the intersection point? Would there be any further chaos? Why not? The intersection point is a fixed point and would capture all future orbits ending the chaos. However, this is a chaotic system, so that will never happen.
[55] For the same reason as the fixed point. If an orbit ever retraced itself, then there would be a predictable periodicity. In other words, the diagram would repeat itself and there would be no more chaos.
[56] And if you believe that, I have a bridge in New York I would be interested in selling…

data! But is this always the case? Can computers make errors? You bet they can! Computers lie all the time. It is simply not possible to represent all real numbers as computer values. We can usually work with 15 or 16 digits of precision (if we are lucky) and no more. This means that the computer real number line is "holey". Nature has no such limitations. When an event occurs in nature, the precision of the "calculations" is infinite. Because of this, we can never completely represent a natural event within the confines of a computer simulation. There will always be some error involved. This may not seem too important… after all, the computer can represent values to 15 or 16 digits so the error will be quite small, right? Wrong. It turns out that most of nature is chaotic. In chaos, the outcome is extremely sensitive to the initial conditions and even vanishingly small errors "feedback" to the extent that the resulting noise overwhelms our attempts to accurately portray nature.

In the previous exercises, what would happen if the graph appeared to retrace itself or strike the intersection point of the logistic parabola and the **y = x** reference line? Would this mean that the model was wrong and the math incorrect? Not at all! The pixels on the display window are not geometric points. Screen pixels are finite in size and there are an infinite number of points between any two pixels. Just as with the number line, we can never display all geometric points on a computer window. So it would be quite possible for a line to appear to intersect a point, when actually it does not. Try to keep this in mind if your graph doesn't seem to do what you expect or if the geometry doesn't seem correct. Of course, the fault may still be human error.

Section 6.6 Predator-prey Relationships

The logistic equation is a simple model[57] of a single population within a habitat. We can expand this model to include populations which are dependent on one another, such as a predator-prey relationship. The Lotka-Volterra predator-prey model[58] was presented as a system of two differential equations as follows:

$$\frac{dx}{dt} = ax - bxy \qquad \text{and} \qquad \frac{dy}{dt} = -cy + exy$$

You probably haven't studied differential equations, so I'll attempt to interpret the meaning of these. Let's define **x** to be the prey population and **y** the predator population. The first equation says something to the effect of "the change in **x** (that's the *dx* term) with respect to time (the *dt* term) is determined by some parameter **a** times the previous **x** population minus some parameter **b** times the interaction between the previous prey and predator populations (**x*y**)". Specifically, we would reason that that change in the prey population with respect to time (**dx/dt**) is affected by how many of these organisms are available to reproduce (**a*x**) and the negative impact of the interaction between predators and prey (**-b*x**). The predator population equation can be translated in a similar fashion. The important thing to realize is that both equations are "wired" together so that they interact with each other as do the populations they model.

How do we translate these differential equations into a form usable by Ruby? The equations assume continuous time, but we can't program continuous time in any computer language.[59] We must take some liberties and rewrite the equations as follows:

```
x = x + (a*x – b*x*y)*dt
y = y + (-c*y + e*x*y)*dt
```

where **dt** represents a very tiny time increment and both **dx** and **dy** are represented by **x** and **y** respectively. In the equations, the new population values are calculated by adding the old values of **x** and **y** to the change in populations represented by the equations in parentheses multiplied by the time increment. We can now use the equations in Ruby and solve them numerically by iteration.[60] Technically, these equations are now called difference equations and the listing below will demonstrate their graphical behavior.

```
# PyPredPrey.py

from OpenGL.GL import *
from OpenGL.GLU import *
```

[57] With not so simple behavior!
[58] http://mathworld.wolfram.com/Lotka-VolterraEquations.html
[59] As far as I know… which may not be very far.
[60] This method or rewriting differential equations is called Euler's Method. You REALLY need to look up Euler. He was an amazing mathematician of the first order. Much of his work was completed after he was completely blind!

```
from OpenGL.GLUT import *
from random import *
from Numeric import *
import sys

#  Initial values of width and height
width = 400
height = 400

def init():
	# White background
	glClearColor(1.0, 1.0, 1.0, 0.0)

	#  Set the projection matrix... our "view"
	glMatrixMode(GL_PROJECTION)
	glLoadIdentity()

	#  Set the plot window range
	gluOrtho2D(0,10,0,6)

	#  Set the matrix for the object we are drawing
	glMatrixMode(GL_MODELVIEW)
	glLoadIdentity()

def plotpredprey():

	# habitat and population parameters
	a = 0.7
	b = 0.5
	c = 0.3
	e = 0.2

	# time increment
	dt = 0.001

	# initial populations
	x = 0.5
	y = 0.5

	glClear(GL_COLOR_BUFFER_BIT)
	glBegin(GL_POINTS)
	for n in arange(0,10, 0.0001):

		# predator-prey equations
		x = x + (a*x - b*x*y)*dt
		y = y + (-c*y + e*x*y)*dt

		# parametric plot
		#glColor3f(0.0,0.0,0.0)
		#glVertex2f(x,y)

		# prey
```

```
            glColor3f(1.0, 0.0, 0.0)
            glVertex2f(n,x)

            # predator
            glColor3f(0.0, 0.0, 1.0)
            glVertex2f(n,y)

      glEnd()
      glFlush()

def keyboard(key, x, y):
      #  Allows us to quit by pressing 'Esc' or 'q'
      if key == chr(27):
            sys.exit()
      if key == "q":
            sys.exit()

def main():

      glutInit(sys.argv)
      glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
      glutInitWindowPosition(100,100)
      glutInitWindowSize(width,height)
      glutCreateWindow("Predator Prey Simulation")
      glutDisplayFunc(plotpredprey)
      glutKeyboardFunc(keyboard)

      init()
      glutMainLoop()

main()

# End of Program
```

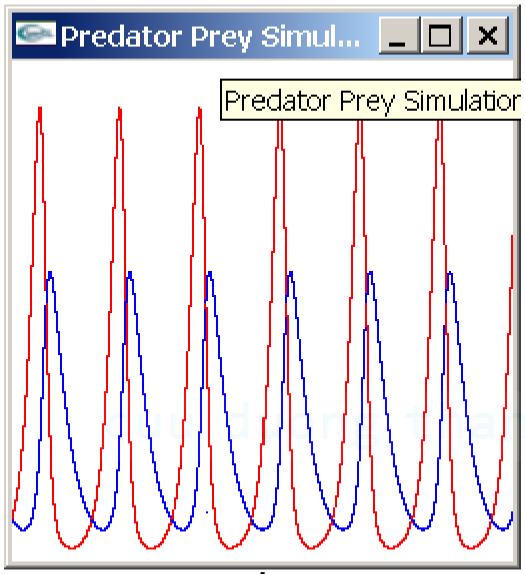
The astute reader may notice the lack of **global** variables within functions. The **global** declaration is needed only if we want to change a variable's value within a function and have that change reflected over the entire program. In this case, we are setting the **width** and **height** variables immediately and won't alter them again during a single run of the program. One note worth mentioning: The difference equations contain the parameter **e** instead of **d** from the sample equations on the first page of this section. This substitution was made to avoid confusion with the **dt** term.

When we run the program, we should see a graphic plot similar to figure "Pred-prey" below. The prey population is plotted in red and the predator population is blue. Notice that the two peaks are related, but they don't coincide. As the prey population reaches its maximum, the predator population begins to grow. As the predator population grows, the prey population declines (they are being eaten!). When the prey population diminishes, the predators begin to starve. When the predators reach a low number, the prey population begins to grow again and the cycle repeats. This is interesting, especially if you like dynamic systems!

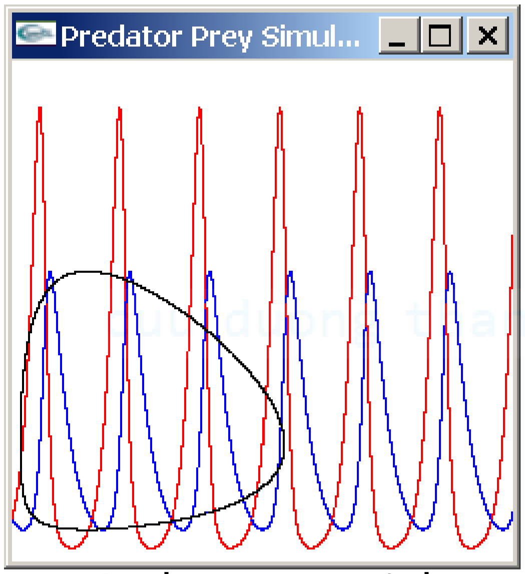
Another interesting way to graph this relationship is parametrically. Uncomment the following **`glColor3f`** and **`glVertex2f`** lines in the **`def plotpredprey():`** function (in the parametric plot section as shown below) and see what happens.

```
# parametric plot
glColor3f(0.0,0.0,0.0)
glVertex2f(x,y)
```

This additional plot (in black) is superimposed on the previous plot as pictured in figure **`Pred-prey parametric`** below.



`Pred-prey`



`Pred-prey parametric`

What does this new plot mean? We are plotting both predator and prey populations with respect to each other. Each point on the “oval” plot represents a single predator-prey ordered pair or relationship, with the prey value on the x-axis and the predator value on the y-axis. Trace the oval plot with your finger and see how it relates to the previous x,y plot of population values. When the prey population is at its highest level (the extreme right side of the oval), the predator population is below 50% of its maximum. When the predator population is at its maximum (the extreme top of the oval), the prey population is very low (the extreme top of the oval is toward the left side of the graph… where the prey population is low). Remember, the x axis still represents prey and the y-axis the predator populations!

This section concludes chapter 6. In the next chapter we’ll explore some 2D strange attractors and present some 2D fractal images that are considered “classic” in both beauty and significance.

Exercises

1) The Lotka-Volterra equations are not chaotic as they are presented. Experiment with the initial population values and see if these have an effect on the plot. Do you notice any changes in the resulting graph?

2) Now experiment with the habitat and population parameters. You may have to adjust the **`gluOrtho2D`** values if the graph becomes too small or too large. Can you induce chaos? That is your goal in this exercise. Can you create a predator-prey model that appears to fluctuate randomly, yet stays within certain limiting values? You don’t want either population to run away to infinity, nor do you want either population to reach extinction (zero). If you are able to achieve chaos, share your model with your classmates and teacher!

3) You can experiment with time by changing the value for **`dt`**. Try **`dt = 0.01`**, followed by **`dt = 0.0001`**. The smaller the "time-slice", the slower the program execution. Why? Also experiment with the increment size in the **`for n in arange(0,10, 0.0001):`** statement. Try **`0.001`** and then try **`0.00001`** and see what happens. You can also try combinations of changes in **`dt`** and the loop increment. However, if you are experimenting to see the effect that individual changes have on a program, you should only change ONE parameter at a time. Why?

4) For an interesting experience, use **`random()`** to randomize the parameters. Here's how:

```
a = random()
b = random()
c = random()
e = random()
```

You can also randomize the initial population values for **`x`** and **`y`**. The **`random()`** command (which is available through the **`from random import *`** statement) generates a pseudo-random number between 0.0 and 1.0, which is perfect for the parameters in this program. I would recommend that you print the parameter values using **`print a`**, **`print b`**, etc. after the **`random`** statements in the event that you get an interesting plot and you want to study it further. Do you see any chaotic behavior with any particular set of parameter values? Chaotic behavior would give the <u>appearance</u> of randomness among the population peaks and valleys rather than a "nice" regular pattern of spikes. The parametric plot of chaotic behavior might be interesting. If you come up with something that appears unusual, let your teacher and classmates see the result!

Far from being simply an interesting command to play with, the pseudo-random numbers generated by **`random()`** are important in many branches of math and statistics… and of course, in game play.

5) What happens when you experiment with the population difference equations themselves? Modify the equations and see what happens. Anything goes! What happens if you substitute trig functions for some of the variables? Your result may not model anything at all in the real world, but perhaps the graph will be interesting.

Chapter 7 Strange Attractors and Beautiful Fractals

In this chapter we will take a closer look at some famous chaotic “strange” attractors and explore the foundations of a few of the most beautiful fractal images ever created. In conjunction with these fractal images, we’ll explore the rudiments of complex arithmetic and learn that imaginary numbers are not imaginary at all! With the exception of a brief interlude when we discuss animation, this chapter will constitute our last formal study of 2D space. First, let’s talk about the weather…

Section 7.1 Lorenz and the Weather

The weather is an intricate part of our daily lives and accurate weather forecasting is both an enormous responsibility and a multi-billion dollar business.[1] It was once thought that with the introduction of computers in the mid-1900’s, weather forecasting would eventually become an exact science (or nearly so!). If we could only bring enough computing power to bear on the problem of forecasting, we could predict with arbitrary certainty the weather many days, or even weeks, in advance. Imagine what this would mean! We would no longer wonder whether[2] or not we should take an umbrella with us to school or work that day. We would know if the game will be played next Sunday afternoon or if we should avoid the 3-hour round trip to the stadium. Farmers could plan strategies for drought or wet conditions during the current growing season. Insurance companies could advise potential home owners to avoid areas slated to be hit by severe storms, tornadoes, or hurricanes. But in 1963, all these hopes of a perfect weather forecasting system were dashed, although few recognized it at the time.

Edward Lorenz was a meteorologist at MIT in the early 1960's.[3] He had a computer, which was not a common thing back in those days. It was hideously slow compared to the machine you are using now,[4] but it did work most of the time. Lorenz used his computer to model the weather and his model was based on three (now famous) differential equations he distilled from a more complex system. The three equations are:

$$\frac{dx}{dt} = -10x + 10y$$

$$\frac{dy}{dt} = 28x - y - xz$$

[1] Many of the most watched programs on television deal with the weather and some of the most popular segments on news shows are the weather forecasts. As I write this, hurricane Katrina is a recent memory and hurricane Rita is bearing down on Texas, it’s exact path uncertain.. Inclement weather has a profound and far-reaching effect on our economy and our daily lives!
[2] Pun intended.
[3] Stewart, Ian (1992). "Does God Play Dice?", Cambridge, MA: Blackwell (pp. 133-144).
[4] The computer was a Royal McBee and had a clock speed of about 1 Hz. Compare that to a typical desktop or laptop computer with a 1 gHz cpu. Your computer is about a billion times faster than Lorenz's!

$$\frac{dz}{dt} = -\frac{8}{3}z + xy$$

The three ***x***, ***y***, and ***z*** equations are dependent on each other and they calculate the change in each variable with respect to time.[5] The key to these equations (and what makes them impossible to solve analytically) is the ***xz*** and the ***xy*** terms in the second and third equations respectively. The only way to effectively understand the behavior of this system of equations was to let a computer solve the equations numerically[6] and that is exactly what Lorenz was doing when he made his monumental discovery.

One day he was running his weather simulation and he wanted to know how a particular solution behaved over a longer period of time. Since his computer was so slow, he decided not to start over at the beginning, but instead, simply entered some numbers from the middle of the run and started the simulation from that point. What he expected to happen (and by all rights, what SHOULD have happened) was that his Royal McBee computer would simply repeat the original run exactly from the new starting point and then go on from there. Lorenz took a coffee break and returned to the printout, expecting to see a duplication of the second half of numbers followed by a new weather pattern. What he saw on the paper was nothing short of a mystery. The numbers did repeat themselves for awhile, but then they slowly diverged until the computer was predicting a completely different weather pattern. How could this be? How could the same equations produce a different result? At first, Lorenz suspected a computer glitch, but then when the computer was found to be running correctly, he realized that something deep was occurring within the equations themselves. He saw that instead of entering the numbers exactly as they were on the printout, he had rounded them to 3 decimal places instead of the usual 6. This seemed reasonable, since everyone knew at the time that small differences in input led to small differences in output… in other words, it shouldn't have made much difference whether the starting values were 3 or 6 decimal places as long as they represented close to the same value. Well, everyone was wrong. It DOES make a difference! Small, even vanishingly small, differences in input can lead to enormous changes in output. Once Lorenz realized this fact, he knew that long-range weather forecasting would be forever out of reach.

All official weather forecasts are based on computer models. These models use systems of equations that rely on data received from various weather stations. This input is limited to a relatively few weather stations scattered worldwide and the measured weather data is limited to a few decimal places depending on the instrument used to collect the data. Nature, however, is not limited to any specific number of decimal places and weather occurs at every point in our atmosphere. Based on Lorenz's discovery, we now know that the outcome of any computer weather model is extremely dependent on the data the model receives. Even if we improve our data collection and instrumentation, we can never hope to achieve accurate prediction more than a few days in advance. This extreme sensitivity to initial conditions has been

---

[5] The three distinct x, y and z variables also imply 3 dimensional behavior, but we will view only 2 of these dimensions at a time in this chapter. We'll explore the full 3D behavior of the Lorenz system later in the text. Notice there is an xy term in the z equation, a y and xz term in the y equation and a y term in the x equation. The equations are "wired" together!
[6] No graphics at all… Lorenz's primitive computer produced only numerical results.

termed the “Butterfly Effect” and is a hallmark of chaotic behavior.[7] The flapping of a butterfly’s wings in Hong Kong leads to a tornado (or not) in Kansas. Thank you, Edward Lorenz…

The following program will attempt to duplicate (as best as we can) Lorenz’s famous computer run. We will have the advantage of graphics, but hopefully we can catch a glimpse of what he saw on his printout and understand the insight he achieved based on his computer model. Here is the program listing:

```
# PyLorenz.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from random import *
from Numeric import *
import sys

# Globals for window width and height
global width
global height

#  Initial values of width and height
width = 500
height = 500

def init():

     # White background
     glClearColor(1.0, 1.0, 1.0, 0.0)

     #  Set the projection matrix... our "view"
     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     #  Set the plot window range
     #  This will coincide with the for... arange loops
     gluOrtho2D(-30.0, 30.0, -30.0, 30.0)

     #  Set the matrix for the object we are drawing
     glMatrixMode(GL_MODELVIEW)
     glLoadIdentity()

def plotlorenz():
     glClear(GL_COLOR_BUFFER_BIT)

     # Enlarge the points for better visibility
```

[7] Remember that chaotic behavior was loosely defined in the last chapter as apparently random behavior produced from deterministic or non-random equations. There are much more precise mathematical definitions to be sure!

```
        glPointSize(2.0)

        # The blue plot is the original plot
        # Note the values for x,y, and z
        x = 0.50002
        y = 0.50002
        z = 0.50002
        dt = 0.0005
        glColor3f(0.0,0.0,1.0)
        glBegin(GL_POINTS)

        # the range is the horizontal width of the window
        for n in arange(-30,30, 0.0005):

                # Lorenz's equations
                x = x + (-10*x + 10*y)*dt
                y = y + (28*x - y - x*z)*dt
                z = z + (-2.66667*z + x*y)*dt

                glVertex2f(n,y)
        glEnd()

        # The second plot in red is the truncated plot
        # Note the small difference in starting x,y,z values!
        x = 0.50000
        y = 0.50000
        z = 0.50000
        dt = 0.0005
        glColor3f(1.0,0.0,0.0)
        glBegin(GL_POINTS)
        for n in arange(-30,30, 0.0005):
                x = x + (-10*x + 10*y)*dt
                y = y + (28*x - y - x*z)*dt
                z = z + (-2.66667*z + x*y)*dt

                glVertex2f(n,y)
        glEnd()

        glFlush()

def keyboard(key, x, y):
        #  Allows us to quit by pressing 'Esc' or 'q'
        if key == chr(27):
                sys.exit()
        if key == "q":
                sys.exit()

def main():
        global width
        global height

        glutInit(sys.argv)
        glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
        glutInitWindowPosition(100,100)
        glutInitWindowSize(width,height)
        glutCreateWindow("Lorenz")
        glutDisplayFunc(plotlorenz)
        glutKeyboardFunc(keyboard)

        init()
        glutMainLoop()

main()

# End Program
```

When you run this program, you'll notice that the red plot and the blue plot coincide for a short distance, with the red overwriting the blue (red is the second plot color). But then the two colors begin to diverge about one-third of the way across the window. By the time the end of the plot is reached, both systems are following completely different paths even though they are using the exact same equations and differ in starting values by only 0.00002! Figure 7.1 below illustrates this strange, but very interesting behavior.

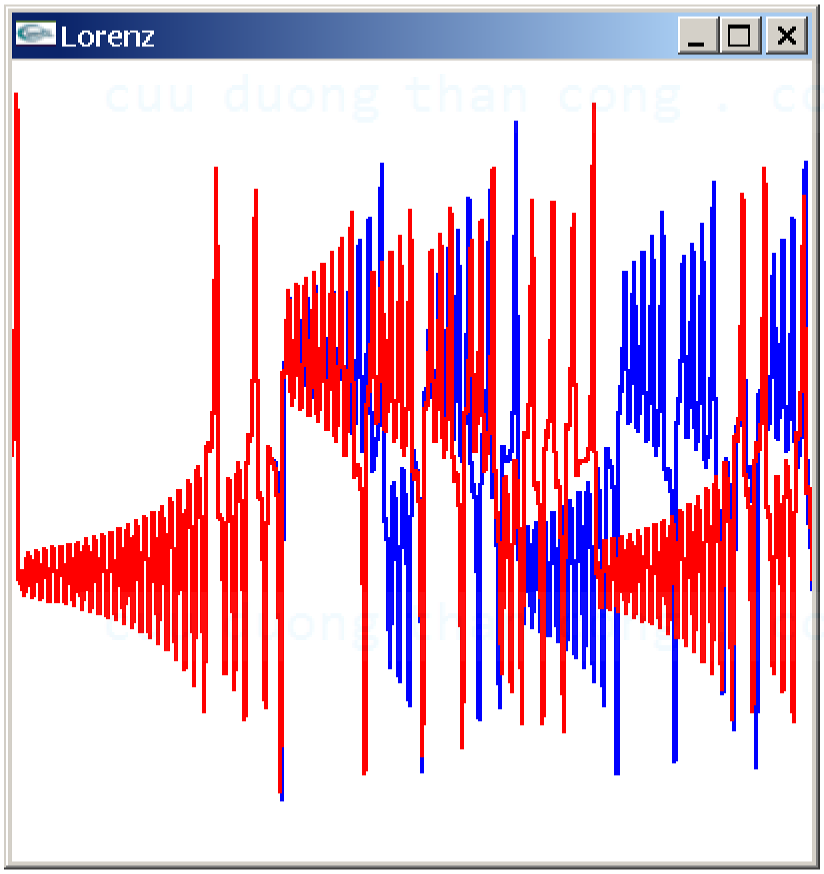


Figure 7.1

What this means is that with any differences or errors in initial weather measurements, no matter how small, completely different weather forecasts will result over a relatively short time. Since all measurements contain some amount of error,[8] it is effectively impossible to predict the weather for any significant time in the future. In fact, we can get two completely different computer weather forecasts for the same place at the same time simply by using data (temperature, pressure, humidity, wind speed, etc.) that differ only in the 3rd or 4th decimal place. Compound this with the knowledge that we don't even know what the weather measurements are between weather stations and you can see how startling this revelation was to a meteorologist such as Lorenz.[9]

The three equations in **`def plotlorenz():`** are the heart of the program and are indented in the **`for`** loop.

```
        for n in arange(-30,30, 0.0005):
             x = x + (-10*x + 10*y)*dt
             y = y + (28*x - y - x*z)*dt
             z = z + (-2.66667*z + x*y)*dt
```

We are letting **`n`** range from **`-30.0`** to **`30.0`**, stepping by **`0.0005`**. There is no magic in choosing **`n`** as a variable name; it could have been something else as long as we avoid using the variables that we are calculating in the loop (why?). In all three equations we are setting the new values for **`x`**, **`y`**, and **`z`** equal to their previous values plus the product of the Lorenz equations and the time increment **`dt`**. Notice how each equation depends on the results of the other equations in order to calculate new values. If you consider that we are calculating a position for a point in space,[10] then the equations become similar to the familiar (hopefully) **`distance = current position + rate*time`** that you learned in science. The **`rate`** for each variable is simply the Lorenz equation for that variable. We plotted **`n`** versus **`x`** in figure 7.1, but you can just as easily plot **`n`** versus **`y`** or **`z`** with the same effect. Try it! Consider **`n`** to represent the passage of time in each plot.

Now we are going to make some changes to the **pylorenz.py** program and view the resulting graphic in a more classic form. Change the **`def plotlorenz():`** subroutine as follows:

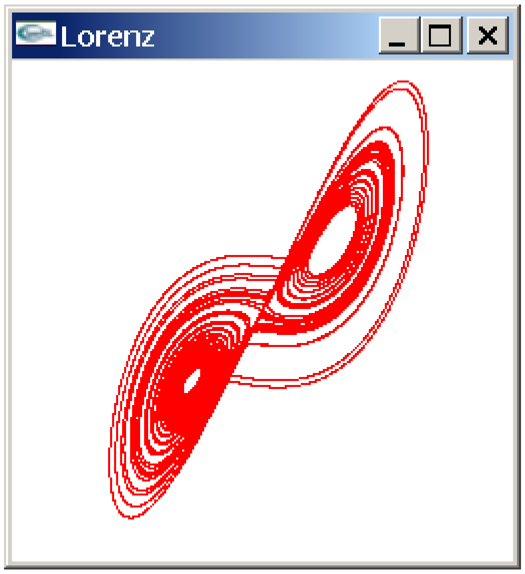
```
def plotlorenz():
     glClear(GL_COLOR_BUFFER_BIT)

     # Initial values for x, y, and z
     x = 0.50000
     y = 0.50000
```

[8] OK, say you measure the temperature of the atmosphere in St. Louis on riverfront and find it to be 37.35^{o} Celcius in July. That's a fairly precise measurement to 4 significant figures. But is the measurement accurate? What about the thousandths decimal place? We don't know anything about it, do we? And how about the temperature one block to the west? Remember that the Lorenz graph we plotted diverged wildly when the difference in starting values was 0.00002!
[9] Even if you placed weather stations every square meter across the globe, the uncertainty of measurement and the unknown weather in the small space between the stations would still result in the unpredictability of the weather over anything other than a very short time period.
[10] This **`x`**, **`y`**, and **`z`** position represents the condition of the system at any point in time.

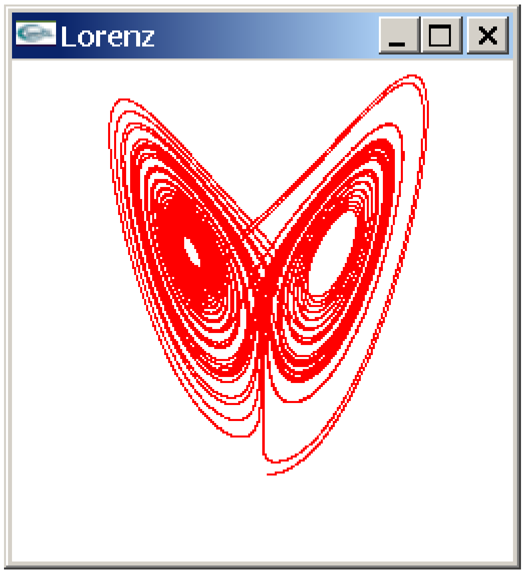
```
    z = 0.50000
    dt = 0.0005

    glColor3f(1.0,0.0,0.0)
    glBegin(GL_POINTS)
    for n in arange(-30,30, 0.0005):
         x = x + (-10*x + 10*y)*dt
         y = y + (28*x - y - x*z)*dt
         z = z + (-2.66667*z + x*y)*dt

         # Plot x versus y
         glVertex2f(x,y)
    glEnd()

    glFlush()

    # End function
```

Essentially we've simplified the code by removing one of the plotting sections and changing the **`glVertex2f`** statement to plot **`x`** versus **`y`** rather than **`n`** versus **`y`**. This is a parametric plot and the result is shown in figure **`x vs. y`** below. This figure is a shadow or projection of the 3D Lorenz attractor onto the **`x-y`** coordinate plane. In 3D space, we have a z-axis which comes out of the monitor toward us and goes into the monitor away from us. Since we are not viewing a 3D screen at this point, all we can do is project the 3D object on one of the 3 planes represented by the 3D coordinate system. This is really no different than viewing a normal picture in a magazine where you see a 3D object (such as a human face) projected onto a flat page. Note: We could have used a different loop structure since we no longer need the variable **`n`** to plot the Lorenz attractor. Can you think of a different loop method that would work here?

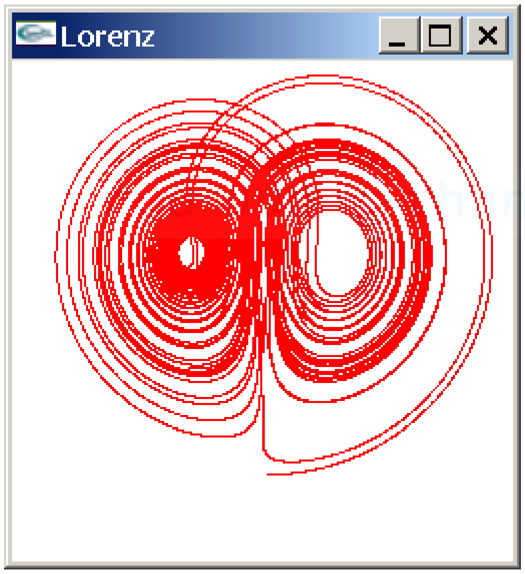
Each point on the projected attractor displays a particular set of atmospheric conditions represented by the values for **`x`**, **`y`**, and **`z`**. These atmospheric conditions continuously "flow" through 3D space as time passes. We can also look at projections of the attractor on the **`x-z`** and **`y-z`** planes as shown in figures **`x vs. z`** and **`y vs. z`** below. Both the **`x vs. z`** and **`y vs. z`** plots have a correction factor in the **`glVertex2f`** statement. This correction factor lowers the plot so that we can see it properly in the window. To program this correction, use **`glVertex2f(x,z-20)`** and **`glVertex2f(y,z-20)`** respectively.



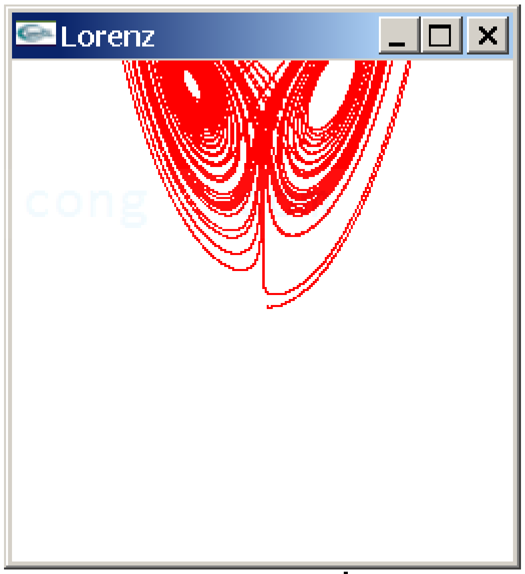
x vs. y



x vs. z



y vs. z



x vs. z orig

The **`x vs. y`** plot is a view of the Lorenz attractor from along the positive **`z`**-axis, while the other plots are "side" views along the **`y`** and **`x`** axes respectively. The view from the **`z-axis`** is centered properly in the window, but the side views (if not corrected) show an attractor that is elevated above the **`x-y`** reference coordinate plane. The **`z-20`** correction we applied in the **`glVertex2f`** statement lowers the plot in each window so that we can see the full shape of the attractor. To see this lowering effect, change the **`z-20`** parameter to simply **`z`** in each statement. You can see the uncorrected original form in the figure **`x vs. z orig`**. I like the corrected views much better! If you are taking CAD drafting, these pictures are orthographic[11] projections of the 3D Lorenz attractor. Using your mental imagery skills, see if you can picture what the Lorenz

[11] In an orthographic projection, there is no foreshortening with distance.

attractor would look like in 3D space. We will revisit this attractor later in the text when we encounter 3D graphics and animation.

The Lorenz attractor is called a “strange attractor”. A strange attractor is an attractor[12] that is not simply a fixed point or a simple shape like a circle. Strange attractors may or may not be visually interesting like the Lorenz attractor, but they tend to signify very complex and often mysterious dynamics. We have already encountered strange attractors in the last chapter with the Sierpinski Gasket and Barnsley Fern and we’ll see a few more examples in the next section.

Exercises

1) You should Google for Lorenz, Lorenz Attractor, and Butterfly Attractor and see what you can find. This is a classic computer simulation and examples can be found written in nearly every computer language with graphics capabilities.

2) Explore different starting values for **`x`**, **`y`**, and **`z`** and see if these values make any noticeable difference in the shape of the attractor. Use the parametric plots of the “Butterfly Attractor” rather than the line graph plot in the beginning of this section. If you notice no discernible difference, what does this say about the definition or nature of an attractor?

3) We can use the varying values of **`x`**, **`y`**, and **`z`** to color the graph with the **`glColor3f`** statement. We have used the **`glColor3f`** statement as a function of variables in the previous chapters, so let’s try it here as well. Make the following changes in the **`plotlorenz`** function:

   ```
   glColor3f(x/8.0,0.20,0.20)
   glVertex2f(y,z-20)
   ```

   This plot is a **`y-z`** projection of the Lorenz attractor. We are colorizing the position of the **`x`** parameter by using **`x/8.0`** in the **`glColor3f`** statement. The resulting graph hints at the 3D nature of the Lorenz attractor by showing in red the portion of the graph that is closest to the observer. This is illustrated in figure **`Faux-3D`** at the end of the exercises. Experiment with this concept and see if you can come up with other interesting color schemes. Change the projection so that you are plotting **`(x,z-20)`** and **`(x,y)`**. What changes should you make in the **`glColor3f`** statement to achieve the most effective plot colors when you change the projection?

4) Try changing the parameters of the equations themselves. For example, in the **`y = y + (28*x - y - x*z)*dt`** equation, try **`24*x`** instead of **`28*x`**. Try **`25*x`** next. See if you can determine the critical value between 24 and 25 which marks the beginning of chaotic behavior.

[12] An attractor “attracts” all nearby points and orbits to it.

5) Return the equations to their original state and then modify the **`x = x + (-10*x + 10*y)*dt`** equation by using different values in place of -10 and 10. Can you find areas of stability or is chaos present regardless of the values you use?

6) Again, return the equations to their original state and modify the **`z = z + (-2.66667*z + x*y)*dt`** equation by changing 2.66667 to different values. What effect does this have on the resulting plots?

7) Make certain you have a saved version of the original program. Now you should take some time to alter Lorenz's equations. As an example, you may want to change the equation **`x = x + (-10*x + 10*y)*dt`** to something on the order of **`x = x + (-10*x – 4.5*z + 10*y)*dt`**. When you modify the equations, be prepared to alter the size of the graphics window using the **`gluOrtho2D`** command. You may need to enlarge or shrink the plot dimension accordingly. For a more interesting plot, try altering the equation **`z = z + (-2.66667*z + x*y)*dt`** to **`z = z + (-0.5*z + x*x)*dt`**. You should probably change **`glVertex2f(x,z-20)`** to **`glVertex2f(x,z-30)`** to center the plot. You can also change the **`gluOrtho2D`** statement to enlarge the graphic plot window. This equation change is illustrated in figure **`z-change`** at the end of the exercises. The **`glColor3f`** statement as a function of **`x`**, **`y`**, and **`z`** was left intact from exercise 3 to provide some indication of the 3D nature of this plot.

8) Another famous strange attractor similar in equation structure to the Lorenz system is the Roessler attractor. The Roessler attractor is also a 3D attractor, but we are going to project it in 2 dimensions similar to the Lorenz attractor. We'll explore the true 3D nature of both attractors later. To create the Roessler attractor, we'll change the **`plotlorenz`** function to **`plotroessler`** as follows:

```
def plotroessler():
   glClear(GL_COLOR_BUFFER_BIT)

   # Initial values
   x = 1.0
   y = 1.0
   z = 1.0
   dt = 0.0005

   glColor3f(1.0,0.0,0.0)
   glBegin(GL_POINTS)

   for n in arange(-90.0, 90.0, 0.0001):
         x = x - (y + z)*dt
         y = y + (x + 0.2*y)*dt
         z = z + (0.2 + x*z - 5.7*z)*dt

         glColor3f(sin(z),cos(x),sin(y))
         glVertex2f(x,z-10)

   glEnd()
   glFlush()
```

```
# End Function
```

Don't forget to change the **glutDisplayFunc** in the **def main():** function to reflect the change in the display function name! Change the **gluOrtho2D** function to **gluOrtho2D(-15.0, 15.0, -15.0, 15.0)** to shrink the viewing window. The Roessler attractor generated with the above code is shown in the figure **Roessler x-z**. You should also try projecting the Roessler attractor on the **y-z** plane using **glVertex2f(y, z-10)** and the **x-y** plane using **glVertex2f(x,y)**. The x-y plane projection is shown in figure **Roessler x-y** at the end of the exercises. The color scheme is arbitrary and may be changed to fit your tastes. Notice again that the **glColor3f** statement is written as a function of the **x**, **y**, and **z** values.

Both the Lorenz and Roessler strange attractors are similar in that they trace the continuous path or orbit of a point in 3D space subject to a set of 3 differential equations. In the next section we will study 2D strange attractors that are visualized as a slice or cross-section of a more complex dynamic system.

9) An additional twist to the coloration of either the Lorenz or the Roessler attractors can be accomplished by using the distance formula from coordinate geometry. As an example, you can use:

```
glColor3f(sqrt(x*x+y*y)/15,sqrt(x*x+z*z)/15,sqrt(y*y+z*z)/15)
```

and color each point based on the distance of that point from the origin. The division by 15 is based on the window dimensions specified in the **gluOrtho2D** command. You can also provide coloration based on a distance from an arbitrary point by using the equation for a circle. For example, you could try:

```
glColor3f(sqrt((x+1)**2 + (y-3)**2)/15 ,0.0 ,0.0)
```

to provide a red shading effect based on the distance from the point **x = -1**, **y = 3**. You can experiment with this effect using **y-z** coordinates, **x-z** coordinates or all 3 coordinates at once. The distance formula works in 3D too!

10) The Rikitake attractor[13] is an interesting fractal in both form and function. See if you can write most of this program on your own with the following hints. The equations are relatively simple:

```
x = x + (-2*x + z*y)*dt
y = y + (-2*y + (z - 5)*x)*dt
z = z + (1 - x*y)*dt
```

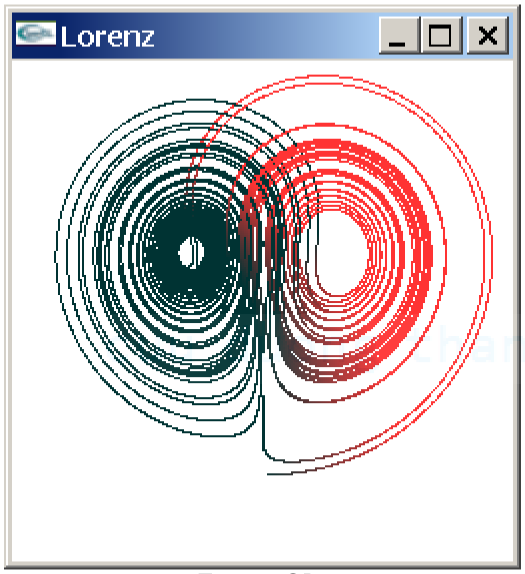
The axis ranges in **gluOrtho2D** are as follows:

```
gluOrtho2D(-7.5, 7.5, -7.5, 7.5)
```

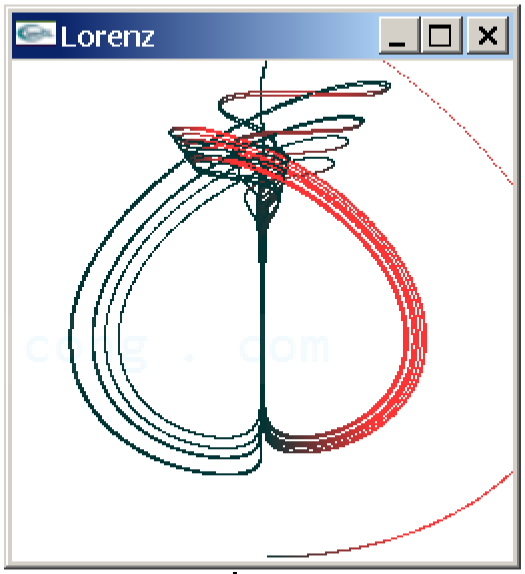
[13] Look up the Rikitake attractor online. It isn't as "famous" as the Lorenz or Roessler attractors.

The initial values of **x**, **y**, and **z** should equal **1.0** and let **dt = 0.001**. In addition, modify **glVertex2f** to **glVertex2f(x, z-5)**. Example plots are found in figures **Rikitake x, z-5** and **Rikitake y, z-5** on page 160. **Rikitake y, z-5** uses **gluOrtho2D(-5.0, 5.0, -5.0, 5.0)** for axis ranges and **glVertex2f(y, z-5)** to properly display the points. The Rikitake attractor is similar to the Lorenz attractor in appearance, but the dynamics in the equations are different. Feel free to change the coloration scheme. These attractors will be more "attractive" in 3D a bit later!

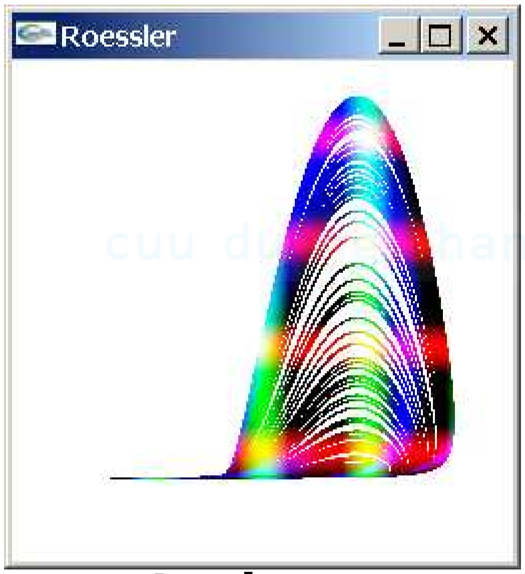
11) Look up the Rainey System attractor and see if you can plot this simple set of equations. If you are successful, do you think the system is chaotic? This exercise will be challenging! You may have to experiment a bit to get an interesting plot.



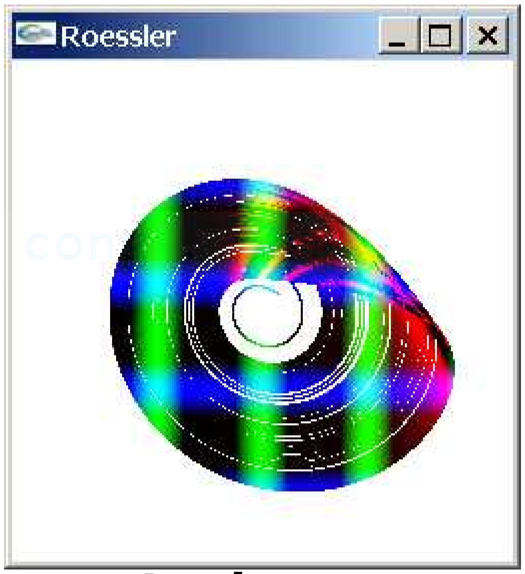
Faux-3D



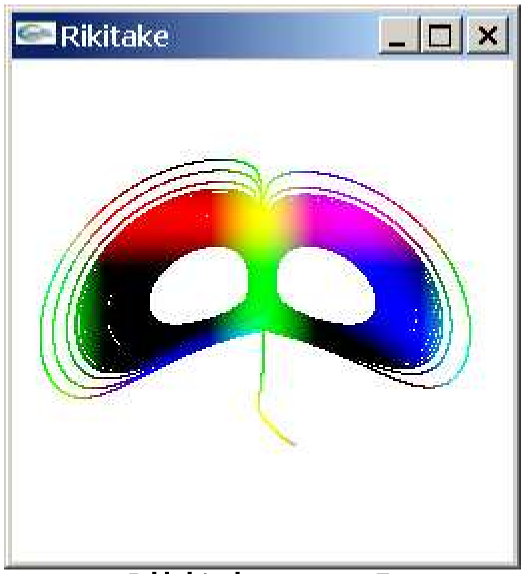
z-change



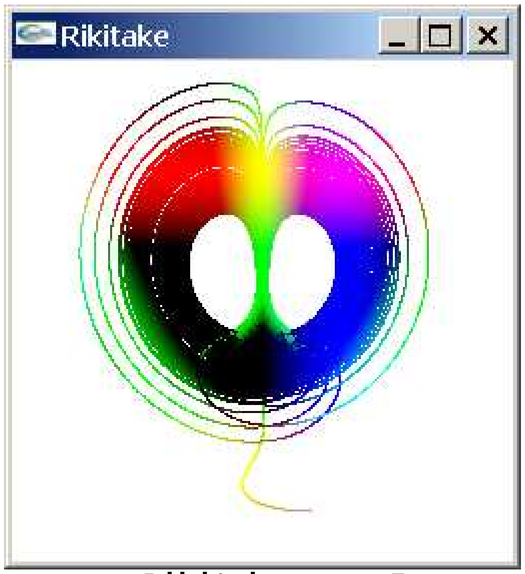
Roessler x-z



Roessler x-y



Rikitake x, z-5



Rikitake y, z-5

Section 7.2 Phase Portraits and Paint Swirls

It should come as no surprise that Time magazine chose Isaac Newton as the 2nd most influential person of the last millennium, behind only Gutenberg. After all, Newton discovered the basic laws that govern the motion of the stars and planets themselves. We still use these laws of motion to send orbiters to Mars and spacecraft to visit the giant planets at the edge of our solar system and beyond. It was thought that by using Newton's laws, we could, with enough information, predict the entire future of the universe with exquisite precision. Fast forward from Newton to the present time. Imagine that on the desk in front of you is a pendulum. This pendulum is ideal in that it does not suffer from friction. Once in motion, it will continue to oscillate forever (or until you get tired of watching it swing back and forth). Now let's also imagine that the pendulum bob is a powerful magnet and on the desk in front of you, beneath the pendulum, are three equally powerful magnets at the vertices of an equilateral triangle. The table magnets are of the same polarity as the pendulum bob, so they repel the pendulum. At rest, the pendulum hangs vertically over the center of the triangle. Now displace the pendulum outside the triangle and let it fall toward one of the vertices. The repelling forces of the magnets will cause the pendulum to swing and bounce erratically from one point to another… and all the bounces and swinging are governed by Newton's Laws of motion. You would think that it should be easy to calculate and predict the future motion of such a device. And you would be wrong. Not only is it NOT easy, it's impossible. We find ourselves back in the world of chaos once again.

Until the advent of the computer, it was truly difficult to study chaotic motion. Using computer graphics, though, we can qualitatively visualize the dynamics of chaotic behavior and at least gather some indication of the behavior of such systems. What we tend to find is that the long-term behavior of a system that appears to be random actually has some structure to it. There is a constrained or deterministic randomness, which is, of course, the hallmark of chaos. In the last section we looked at the continuous[14] orbits of 3D attractors and such orbits are certainly interesting. Each 3D point in space on the Lorenz attractor, for example, represents 3 conditions of the atmosphere at that point. One of the conditions (the **x** variable) is the convective motion of the atmosphere and the other two variables represent horizontal and vertical temperature gradients. Taken together with the right equation parameters, we get the fascinating Lorenz "Butterfly" as a result of the 3 difference equations. There are other ways to visual chaotic behavior, though. Are you hungry? Think of a doughnut. A mathematical doughnut is called a torus. A torus can be made by taking a circle at the origin, moving the circle to the right a few units, and then have the circle orbit the origin at a fixed radius. It turns out that much chaotic behavior can be thought of as orbits inside a torus or donut. The orbit goes around and around inside the donut, always changing its path a bit so that it never follows the same orbit. Yet the path never leaves the torus or donut, either. We could visualize the whole (hole?) dynamics by looking at the 3D donut/torus,[15] but there is

[14] Not continuous in the strict mathematical sense. Remember that we must simulate the continuous behavior of nature by using time slices or steps in our computer simulations. In the Lorenz and Roessler attractors, time steps were simulated using the variable **dt.**
[15] We are not limited to 3D space. We can have 4D, 5D, or nD space as well. It's just a little more difficult to visualize such dynamics. You can also think of the orbits of asteroids in the asteroid belt. The belt is a rough torus-shaped region of space between Mars and Jupiter. If you followed a single asteroid around its orbit for several years, you would find that when it crossed a

another possibility. Why not cut the donut and look at a cross-section of the paths or orbits? The cross-section of the donut is a circle and when the orbit crosses the circle, it will plot a point. The collection of orbit points in cross-section is called a "phase portrait", "Poincaire Section", or "Poincaire Map" of the dynamical system. What will the collection of orbit points look like over time? That's the $64000 question![16] We'll choose a simple system and see. The system we'll start with is called the Ikeda attractor and it looks a lot like the swirl you see when mixing paint.[17]

```
# PyIkedaAttractor.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from random import *
from Numeric import *
import sys

#  Initial values of width and height
width = 600
height = 600

def init():
      # White background
      glClearColor(1.0, 1.0, 1.0, 0.0)

      # Ugly Purple Plot
      glColor3f(0.45, 0.3, 0.5)

      #  Set the projection matrix... our "view"
      glMatrixMode(GL_PROJECTION)
      glLoadIdentity()

      #  Set the plot window range
      gluOrtho2D(-0.75, 2.0, -2.25, 1.25)

      #  Set the matrix for the object we are drawing
      glMatrixMode(GL_MODELVIEW)
      glLoadIdentity()

def plotikeda():

      # Choose an initial point... any point
      x = 0.5
      y = 0.5
```

particular region of space… a circular cross-section of space, that it would be in a different location each orbit.

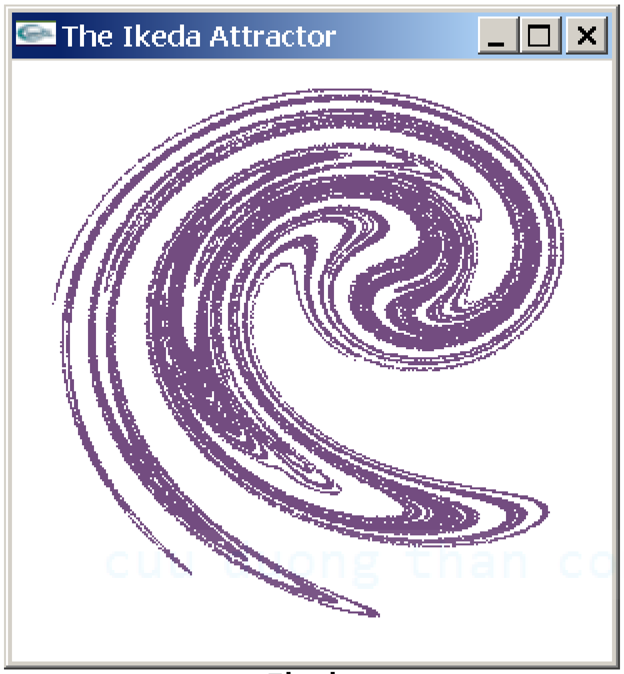
[16] I think this used to be the title of a game show on television. I'm too young to remember such distant happenings, though. From what I understand, the show was "fixed" and was the focus of several investigations.

[17] Mixing is an important concept in chaos theory.

```
    glClear(GL_COLOR_BUFFER_BIT)
    glBegin(GL_POINTS)

    for n in range(0,100000):
        temp = 0.4 - 7.7/(1+x*x + y*y)
        xx = 1 + 0.9*(x*cos(temp) - y*sin(temp))
        y = 0.9*(x*sin(temp) + y*cos(temp))
        x = xx

        #glColor3f(cos(x), sin(y), tan(x))
        glVertex2f(x,y)
    glEnd()
    glFlush()

def keyboard(key, x, y):
    #  Allows us to quit by pressing 'Esc' or 'q'
    if key == chr(27):
        sys.exit()
    if key == "q":
        sys.exit()

def main():
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
    glutInitWindowPosition(100,100)
    glutInitWindowSize(width,height)
    glutCreateWindow("The Ikeda Attractor")
    glutDisplayFunc(plotikeda)
    glutKeyboardFunc(keyboard)

    init()
    glutMainLoop()

main()

# End Program
```

The resulting plot is shown in figure **Ikeda** below. It does indeed look like a swirl of paint. The simplicity of the overall shape of the plot reveals structure in the attractor that would not be apparent from the numerical values generated by the equations for the system. The individual points are plotted, seemingly at random times and locations, yet the shape of the graph is distinct and does not change from run to run. You should be able to tell by now that we are plotting 100000 points (how do you know this?) using the Ikeda system of equations. The equations are a bit more complex than the Roessler or Lorenz, but they are wired together in much the same fashion. You can uncomment the **glColor3f** statement to provide a bit more spice to the graph if you wish. Also notice that the **gluOrtho2D(-0.75, 2.0, -2.25, 1.25)** statement is not centered on the origin and is not "square" even though the graphics window IS square. You might try modifying this to **gluOrtho2D(-2.25, 2.25, -2.25, 2.25)** and see how the new windows dimensions alter the view of the attractor.

What does the Ikeda attractor represent? It represents the time evolution of an optical cavity containing a nonlinear dielectric subjected to a periodic string of light pulses.[18] I'm not certain what that means, either, but it sounds important. What we need to understand, though, is that prior to computers we would have had a difficult time analyzing any chaotic system, much less this one. A scientist familiar with the physics represented here would no doubt have an "Aha!" moment when viewing this graph.



Ikeda

Note: You may be wondering about the absence of **`global`** variables? Any variable defined outside a function, such as **`width`** and **`height`** in this program, can be "seen" throughout the entire program so we can automatically assume that such variables are global "read-only" variables. However, if you want to change the <u>value</u> of a global variable in a function, you MUST declare the variable as **`global`** both outside the function and inside the function as we have previously seen. In this program, we do not intend to change the value of **`width`** and **`height`**. We only need to use those values to initialize the window dimensions.

Exercises

1) Google the Ikeda attractor and look at its history and definition.

2) As in other topics we have studied, experiment with the parameters and equations in this program and see if you can find other attractor shapes. Make certain you keep a copy of the original program for reference.

[18] http://www-chaos.umd.edu/misc/attractorpics.html

3) Try plotting more or fewer points and see how the attractor is affected. Also experiment with the color as a function of the variables in the equations.

Section 7.3 Mira (Look?)

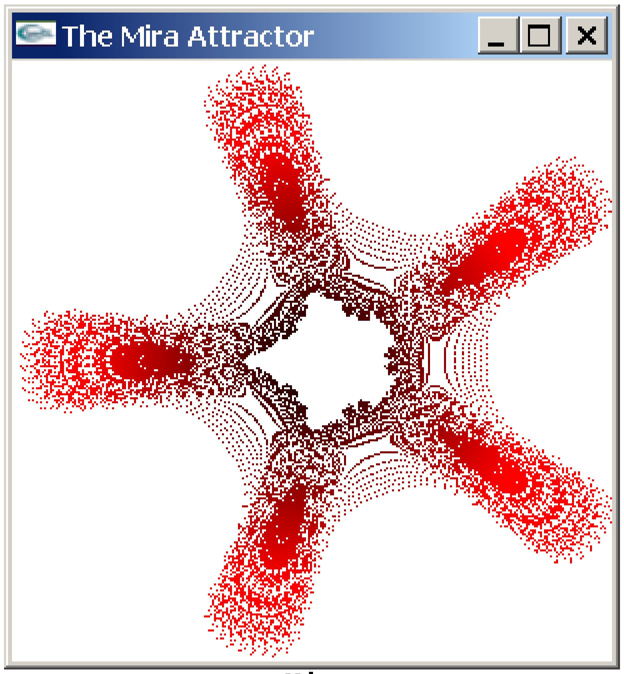
Another attractor that displays enormous sensitivity to the initial values of the parameters is the Gumowski-Mira attractor. We can use most of the skeleton code (remember **pyskel.py**?) from the Ikeda attractor to create the Mira attractor. The changes are as follows. In the **def init():** function:

```
      gluOrtho2D(-20.0, 20.0, -20.0, 20.0)
```

The **plotmira** display function is listed below:

```
def plotmira():
      glClear(GL_COLOR_BUFFER_BIT)

      # Initial values for parameters
      # This attractor is very sensitive
      # To the values for x, y, a, and b
      x = 12
      y = 0
      a = 0.301
      b = 0.9998
      c = 2 - 2*a
      w = a*x + c*x*x/(1+x*x)

      glBegin(GL_POINTS)

      # Plot a significant number of points
      for n in arange(0,100000):
            z = x
            x = b*y + w
            u = x*x
            w = a*x + c*u/(1 + u)
            y = w - z

            # Don't plot anything until we've hit the attractor
            if n > 100:
                        # How does this color statement work?
                        glColor3f(sqrt(x*x + y*y)/15, 0.0, 0.0)
                        glVertex2f(x,y)
      glEnd()
      glFlush()

      # End function
```

You must also remember to change the **glutDisplayFunc()** statement in **def main():** to **glutDisplayFunc(plotmira)** to reflect the renaming of the display function to **def plotmira():**. You should also change the display window caption to something like "The Mira Attractor".[19] The Mira attractor generated by this particular code and set of parameters is displayed in figure **`Mira`** below.



`Mira`

The patterns you can create with the Mira code are amazing and are extremely sensitive to each of the parameters listed. For example, change the initial **`x,y`** point from **`x = 12`** and **`y = 0`** to **`x = 15`** and **`y = 0`** and change parameter **`a`** from **`a = 0.301`** to **`a = 0.7`** and see what happens. Different, right? Now try **`a = 0.107`**. You may want to change the **`gluOrtho2D`** statement to something similar to **`gluOrtho2D(-15.0, 15.0, -15.0, 15.0)`** to shrink the window and enlarge the attractor.[20] This latest version of the Mira attractor is displayed in the figure **`Mira a = 0.107`**. at the end of the exercises.

The equations for the Gumowski-Mira attractor originated in the particle physics experiments conducted at CERN in Switzerland. The orbits of high energy particles such as protons must be stable within the accelerator ring. Gumowski and Mira developed their attractor model to explore the paths these elementary particles take at high energies.[21] The result was apparently useful to physics, but certainly beautiful to those of us who are not professional physicists.

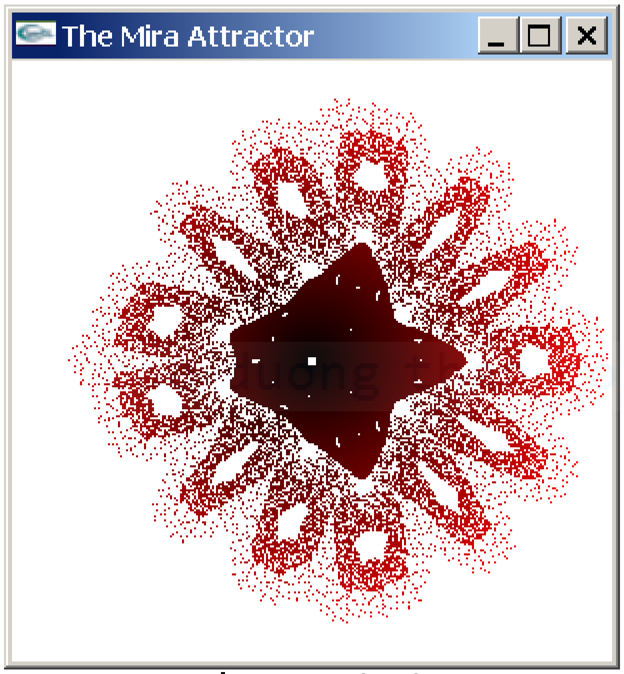
[19] Where and how do you accomplish this task?
[20] Be prepared to do this anytime you need to change the size of a drawing. There are two ways to increase the size of an object: you can simply increase the size of the object or you can shrink the display window. Both work equally well and result in the same effect.
[21] Lauwerier, Hans (1991). "Fractals: Endlessly Repeated Geometrical Figures". Princeton Science Library. pp. 136-137.

Exercises

1) Google the Gumowski-Mira attractor. What is the gingerbread attractor? Can you create the gingerbread attractor on your computer?

2) Continue to explore various starting values for **x** and **y**. Also explore different parameter values for **a** and **b**. Try plotting more or fewer points and see how this affects the resulting graph.

3) Change the equations and see if you can create your own attractor. If so, you can name it after yourself.



Mira a = 0.107

Section 7.4 The 3-Body Problem

When Isaac Newton discovered the laws of motion, he quickly found the solution to the problem of the gravitational effects and resulting paths of two bodies in orbit around each other. From this success, he reasoned that there were similar “easy” solutions for the gravitational motion of any number of objects. However, when others,[22] particularly Henri Poincaire, attempted to find an analytical solution for Newton's equations involving just 3 bodies, the problem was impossible to solve. Poincaire had discovered chaos in a relatively simple orbital model.

As a result of work by Henon,[23] a system of equations was developed to explore the phase portrait or Poincaire map of the 3-body problem. Remember that we can consider a phase portrait to be a cross-section in time of the orbit of a system around the interior of a torus... in other words, a view of the face of a donut slice. The phase portrait plotted in this section shows that far from being completely random, the motion of 3-bodies in space is chaotic, but definitely under the influence of deterministic equations. There is structure in the phase portrait! As in the Mira attractor, we can use much of the code in the Ikeda program as a template for this section. First, modify the **gluOrtho2D** statement as follows:

```
	gluOrtho2D(-1.0, 1.0, -1.0, 1.0)
```

and change the **def plotfunc():** function to:

```
def plot3Body():
	a = 1.16

	glClear(GL_COLOR_BUFFER_BIT)
	glBegin(GL_POINTS)
	for x in arange(0,1.0, 0.05):
		for y in arange(0,1.0, 0.05):
			for i in arange(1,1000):
			  xx = x*cos(a) - (y-x*x)*sin(a)
			  y = x*sin(a) + (y-x*x)*cos(a)
			  x = xx

			  if x > 1.0 or x < -1.0 or y > 1.0 or y < -1.0:
			     break
			  glColor3f(cos(i),sin(i),tan(i))
			  glVertex2f(x,y)
	glEnd()
	glFlush()

	# End Function
```

[22] Stewart, Ian (2002). “Does God Play Dice? The New Mathematics of Chaos”. Blackwell Publishing, Second Edition. pp. 49-63
[23] Ibid pp. 140-142, also Lauwerier pp. 128-133

Make certain you the change the **`glutDisplayFunc()`** statement in **`def main():`** to **`glutDisplayFunc(plot3Body)`** to reflect the renaming of the display function to **`def plot3Body():`**. There are some new concepts in this program that need some explanation. Let's start with the complicated conditional statement in the **`def plot3Body():`** function. In the course of running this program, the values for **`x`** and **`y`** can exceed the limits of the **`glVertex2f`** command and an error will occur. To see this firsthand, comment out the following statements as follows:

```
# if x > 1.0 or x < -1.0 or y > 1.0 or y < -1.0:
#     break
```

The error that is eventually generated stops program execution immediately and we do not see the complete plot of the attractor. The purpose of the **`if`** conditional and subsequent **`break`** statements is to catch the error before it occurs. If we exceed the plot limits we specified in the **`gluOrtho2D(-1.0, 1.0, -1.0, 1.0),`** then we want to skip that particular point. In other words, if **`x`** is greater than 1.0 or **`x`** is less than -1.0 or if **`y`** is greater than 1.0 or **`y`** is less then -1.0, the plot would be outside the screen limits and we couldn't see it at all. Why even try to plot that point? Also, the values generated for **`x`** and **`y`** actually exceed the limitations of the **`glVertex2f`** statement during some calculations. We need to either correct this behavior or catch it so that it doesn't affect program execution. What happens in the conditional statement is that when the problem occurs, we immediately **`break`** out of the current loop and try the next value for **`x`** or **`y`**. This prevents the program error from occurring. Remember this little trick in your own projects when errors occur! **NOTE**: The latest version of Python (2.5) and Numeric did not generate an error in this section of code. However, remember this tip to circumvent errors should they occur in any of your programs. We could also write the if conditional as follows:

```
if abs(x) > 1.0 or abs(y) > 1.0:
     break
```

Why would this work? What is the purpose of the **`abs()`** function?

The next and most important explanation concerns the series of nested loops:

```
for x in arange(0,1.0, 0.05):
     for y in arange(0,1.0, 0.05):
          for i in arange(1,1000):
```

One difference in this program when compared with previous examples is that in previous plots, we've taken a single point (as in the Mira attractor) or a random series of points (as in the Barnsley Fern) as input into our equation system. In this program, we are actually systematically sampling every[24] **`(x,y)`** point in quadrant I of the standard Cartesian coordinate system (stepping by 0.05) and using these points as our initial equation input values. This is the purpose of the nested loop structure; to provide us

[24] Obviously not EVERY point. In this example, we are sampling every 0.05 points from 0.0 to 0.95. The idea is that we are looking at a series of points that cover the coordinate system and running those points through our equations. We'll be doing this again in the fractal sections of this chapter.

with all **(x,y)** ordered pairs within the limits of our loop statements. In the outer loop, we start by assigning **x = 0.0** (how?) and sample each 0.05 units until **x = 0.95**. For each value of **x**, we use the nested **y** loop to sample every **y** value from 0.0 to 0.95, stepping again by 0.05. This will result in the series of coordinates

(0.0, 0.0)
(0.0, 0.05)
(0.0, 0.10)
.
.
(0.0, 0.95)
(0.05, 0.0)
(0.05, 0.05)
(0.05, 0.10)
.
.
(0.90, 0.95)
(0.95, 0.0)
(0.95, 0.05)
.
.
(0.95, 0.90)
(0.95, 0.95)

You can visualize this by imagining a matrix of dots in the graphics window, with each "dot" representing an **(x,y)** ordered pair from the nested loops. Once we've chosen an ordered pair with our **x** and **y** loops, we use the **i** loop to iterate the ordered pair 1000 times in our equations to see what happens. The resulting plot is shown in the Figure **3-Body** at the end of the exercises. Notice the chaotic behavior in the outer fringes of the plot!

You can easily see the effect of the nested loops by simply choosing a single **(x,y)** ordered pair rather than taking a systematic sampling. Alter the **def plot3body():** function as follows:

```
def plot3Body():
     a = 1.16

     # A fixed x,y point
     x = 0.475
     y = 0.455

     glClear(GL_COLOR_BUFFER_BIT)
     glBegin(GL_POINTS)

     #for x in arange(0,1.0,.05):
     #    for y in arange(0,1.0,.05):
     for i in arange(1,1000):
          xx = x*cos(a) - (y-x*x)*sin(a)
          y = x*sin(a) + (y-x*x)*cos(a)
```

```
            x = xx

            if x > 1.0 or x < -1.0 or y > 1.0 or y < -1.0:
                 break
            glColor3f(cos(i),sin(i),tan(i))
            glVertex2f(x,y)
       glEnd()
       glFlush()

       # End Function
```

Pay close attention to the change in the indentations after commenting both outer **`for`** loops! This indentation modification is necessary for this program to run. Remember that Python is very strict about indenting. If you run the program with the changes above, you'll see something like figure **`Fixed Init x-y`** after the exercises. This illustrates the orbit of a single fixed point, the ordered pair (0.475, 0.455). Notice this orbit is NOT chaotic and represents a stable pattern of motion over time. The attractor illustrated in Figure **`3-Body`** is the result of adding ALL of the plots from the systematic sampling of each (x,y) ordered pair in the nested loop structure.

Exercises

1) There are a number of explorations that we can perform with this program. First, make certain that the program listing for the **`def plot3Body():`** function is in its original configuration. Now experiment with changing the **`a`** parameter. Here are some values you can try one at a time. Which values for **`a`** lead to chaotic regimes? Can you predict what values for **`a`** will result in chaos? The results of each of these initial a parameters are found in the appropriate figures at the end of the exercises.

   ```
   a = 1.33
   a = 1.58
   a = 2.0
   a = 2.04
   a = 2.21
   a = 2.71
   ```

   Now try some of your own values for **`a`**. What happens if **`a > 3.0`**? **`a < 1.0`**?

2) Try changing the number of iterations in the **`for i`** loop to something other than **`1000`**. What does the plot look like with **`100`** iterations? How about **`10000`** iterations?

3) In the example program for this section, we started both the **`x`** and **`y`** loops at 0. What happens if you start at 0.5 for both loops? How about 0.75? You may have to adjust the step to get a plot that is pleasing to the eye. Look at exercise 4 for details.

4) What happens if you step by something other than **`0.05`** in the **`for x`** and **`for y`** loops? Try stepping by **`0.1`** and then by **`0.001`**. What is the difference in the plots?

5) The coloration of the 3-Body plot is provided by the **`sin(i)`** function within the **`glColor3f`** statement in **`def plot3Body():`**. Explore different functions (trig or otherwise) for coloration. Try using **`x`**, **`y`**, and **`i`** as indices for any functions you create. Remember that if you want the coloration to vary within the plot, you must provide some variables or parameters that vary. In this program, **`x`**, **`y`**, and **`i`** vary accordingly. **`x`** and **`y`** vary according to the equations and **`i`** varies according to the **`for i`** loop. Invent some functions using these variables for unique color schemes.

6) The GingerBread Man Fractal is interesting in that the plot actually looks humanoid. In order to create the fractal, first modify the **`gluOrtho2D`** function as follows:

```
gluOrtho2D(-8.0, 10.0, -8.0, 10.0)
```

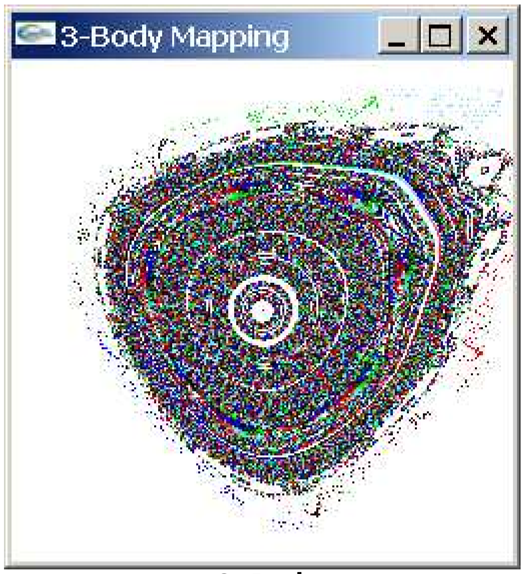
Then make certain your **`def plotfunc():`** function looks like the following:

```
def plotfunc():
     glClear(GL_COLOR_BUFFER_BIT)

     # Initial values for parameters
     # This attractor is very sensitive
     # To the values for x, y, a, and b
     x = -0.1
     y = 0

     glBegin(GL_POINTS)

     # Plot a significant number of points
     for n in arange(0,50000):
          xx = 1 - y + abs(x)
          y = x
          x = xx

          # Don't plot until we've hit the attractor
          if n > 100:
               # How does this color statement work?
               glColor3f(sqrt(x*x+y*y)/15, 0.0, 0.0)
               glVertex2f(x,y)
     glEnd()
     glFlush()
```

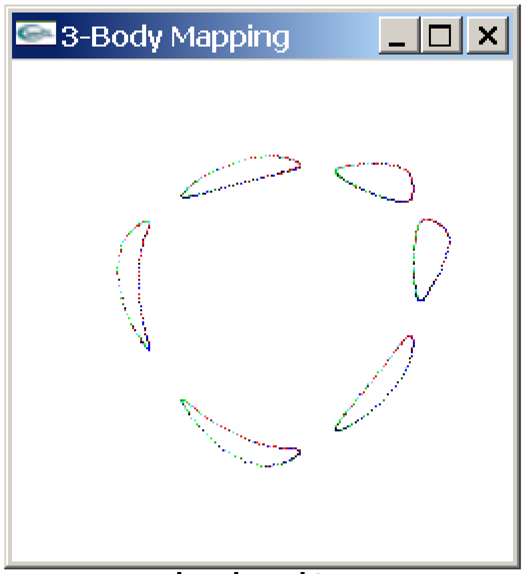
The result can be seen in the **`GingerBread Man`** figure at the end of the exercises. Feel free to experiment with this fractal! Who knows what monstrosities you might create?

7) This exercise is a challenge. Google for the Henon strange attractor. Your assignment is to write a program that displays the Henon strange attractor. You may use the programs we've written in this chapter as models if that helps. Pay particular attention to the equations and starting parameters (a, b, c, etc) needed to plot the Henon attractor. See figure **`Henon`** at the end of the figure set for an example of a

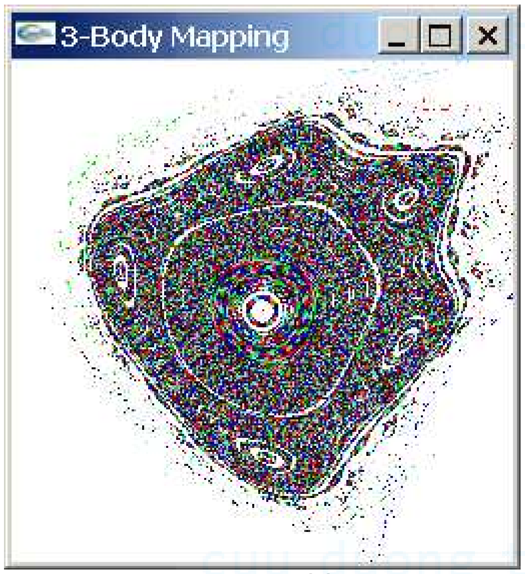
Henon attractor plot. It doesn't appear very impressive, but this is attractor is famous in the history of chaos theory.



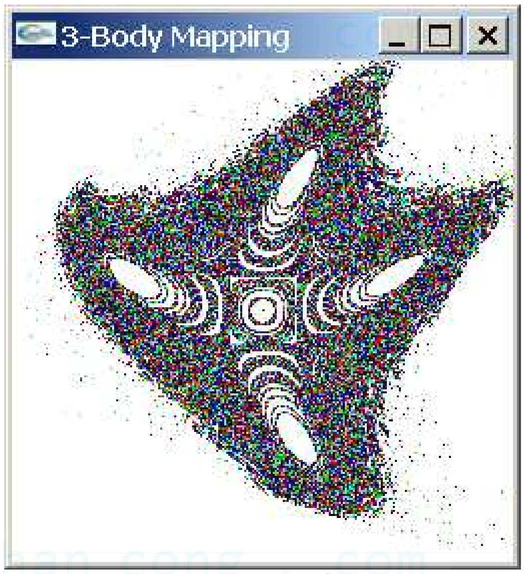
3-Body



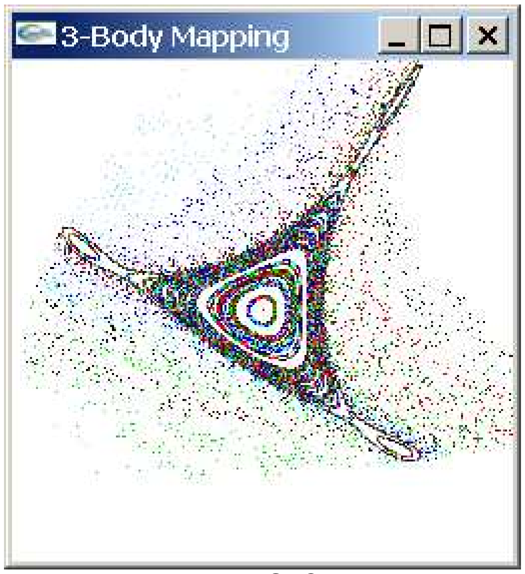
Fixed Init x-y



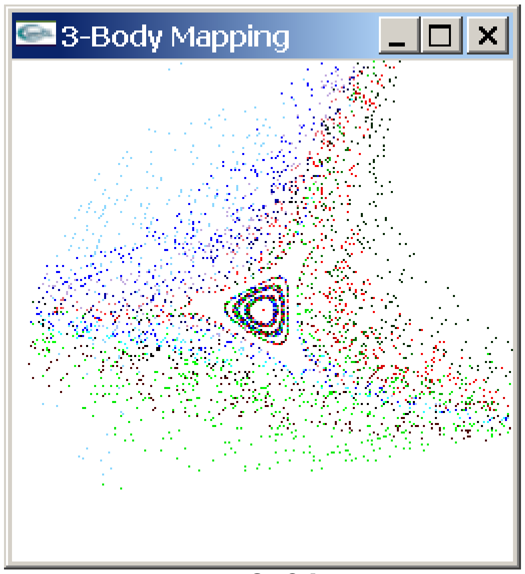
a = 1.33



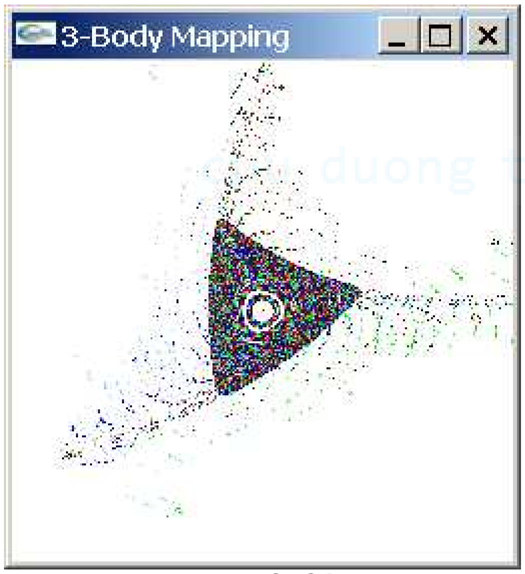
a = 1.58



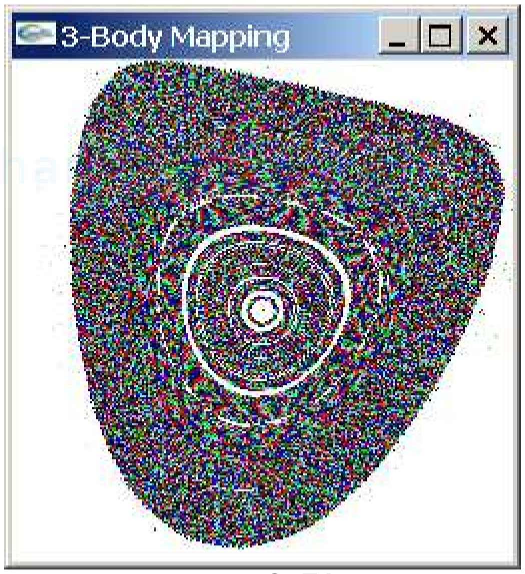
a = 2.0



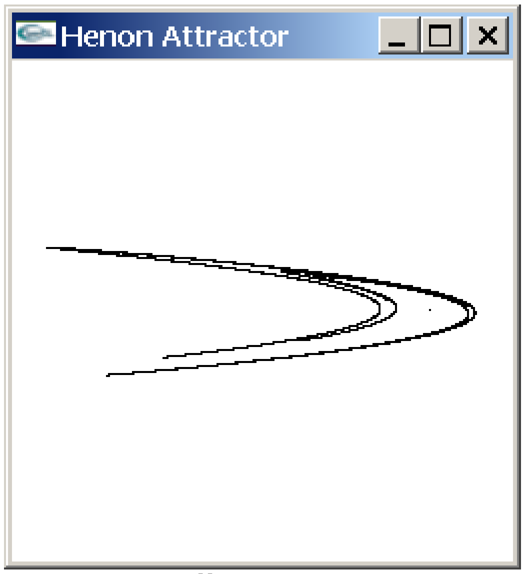
a = 2.04



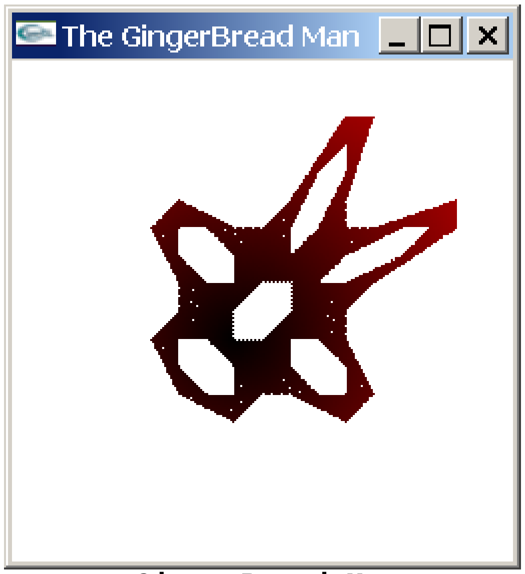
a = 2.21



a = 2.71



Henon



GingerBread Man

Section 7.5 Newton's Method and the Complex Plane

As stated earlier in the text, it is almost impossible to overstate the importance of Isaac Newton in the fields of mathematics and physics. Newton is responsible for the invention (or discovery?) of the calculus and its applications in the sciences.[25] In this section we are going to explore Newton's method for finding the roots of a polynomial equation and we are going to apply the method in the complex plane to generate some remarkable fractal images. Newton's method uses the concept of a derivative to calculate the roots of a polynomial equation via the process of iteration. You already know that you can easily solve equations such as $3x – 7 = 2$. Equations such as $x^2 – 3x + 1 = 0$ may be solved by factoring or by using the quadratic formula. Cubic equations such as $x^3 – 3x^2 + 5x - 1$ are a bit more difficult, but there does exist a cubic formula for this purpose. However, the quartic formula is a nasty thing[26] and the quintic formula (and above) does not exist. So how can we find the roots to such equations? One answer is by Newton's method.

Newton's method is an iterative equation that is represented as follows:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

In order to use Newton's method, we must supply an initial seed or "guess" concerning the root of the equation (this is x_0) and subtract from this seed value the quotient of the function at x_0 divided by the derivative of the function at x_0. This results in a new value x_1, which is then used as input for the next calculation. In most cases the iteration of this equation will converge on one of the roots of the polynomial function you are working with. To find all the roots of a higher order polynomial would require a seed value near each root, but in most cases this can be accomplished, especially if the function has been plotted and the roots approximated. There are some instances where Newton's method does not converge to a root. This can be caused by a poor initial guess or by a function that has no real roots.

An example of Newton's method at work can be seen in the simple square root function. Let's assume that we wish to calculate the square root of 5. We can express this as an equation, $x^2 = 5$ or $x^2 – 5 = 0$. The derivative[27] of $x^2 – 5$ is $2x$, so we can rewrite Newton's equation as:

$$x_1 = x_0 - \frac{{x_0}^2 - 5}{2x_0}$$

[25] Leibniz also invented/discovered the calculus independently of Newton at about the same time.
[26] http://planetmath.org/encyclopedia/QuarticFormula.html
[27] Finding derivatives is a large portion of a first semester calculus class. In this text, I'll supply them for you. You really should take calculus… it's an amazing subject and is the gateway to higher mathematics.

Now let's use this equation to find the square root of 5. First, we'll choose an initial guess of 2 for the square root of 5. Plugging 2 into the equation as follows:

$$x_1 = 2 - \frac{2^2 - 5}{2(2)}$$

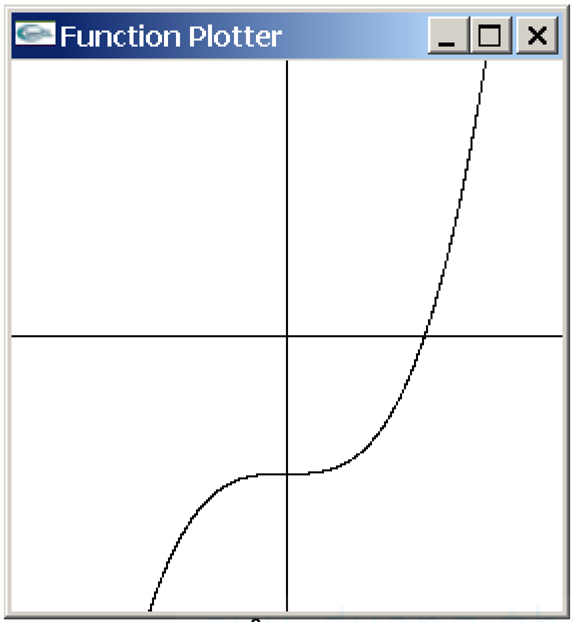
results in a value of 2.25 for x_1. Using 2.25 as the next value for x_0 yields 2.23611111111. Iterating twice more will provide a stable solution of 2.2360679775, which isn't the exact square root of 5,[28] but it's accurate to 10 decimal places! Since $x^2 - 5 = 0$ is a function of degree 2, from your algebra classes you know that it has two roots. In this example, both roots are real and we can find the other root by using -2.00 as the initial seed. After just a few iterations we find a value -2.2360679775 for the second root. When using Newton's method, it helps to know a little bit about the function you are studying[29] so that you can choose initial root values that have a good chance of converging to a stable value.

Some functions, though, do not have real roots or have a combination of real and complex[30] roots. Examples are $x^2 + 1 = 0$, which has no real roots and two complex roots[31] and $x^3 - 1 = 0$ which has one real root and two complex roots. We can directly view the real root of this equation by using the function plotting program from section 5.2. Figure **`x³ – 1 = 0`** on the next page displays this function. Setting **`$axrng = 2.0`** in this plot, we can see that the function crosses the **`x`** axis at 1.0, the only real root of this function. There are also two complex roots for this equation. Can we use Newton's method to visualize these roots? Perhaps we can, but we need to bring in some additional information first. You should be familiar with the concept of a number line. All real numbers can be placed on a number line with zero in the center and extending to negative infinity on the left and positive infinity on the right. Let's expand the number line to include the complex numbers. We'll do this by using the traditional Cartesian x-y coordinate plane and placing the complex numbers on the y axis. Any complex number such as 4 + 3i can be plotted in this new coordinate system by an ordered pair, in this case by (4, 3i). The x-i coordinate system can be called the complex coordinate plane and figure 7.2 illustrates this plane and the (4, 3i) example point.

Doing arithmetic with complex numbers can be a bit tricky. Addition and subtraction is simple and is exactly like the addition and subtraction of regular ordered pairs of numbers, for example (3 + 4i) + (5 - 2i) = (8 + 2i). Multiplication and division are a bit more difficult. First, we need to remember that $i^2 = -1$, $i^3 = -i$, and $i^4 = 1$. Multiplying (3 + 4i) (5 - 2i) yields ($15 + 20i - 6i - 8i^2$), which simplifies to (15 + 14i -8(-1)) = (23 + 14i). With division, we multiply both the complex numerator and the complex denominator by the complex conjugate of the denominator. This produces a real number in the denominator and division then becomes possible. An example would be (3 + 4i) / (5 - 2i). Multiplying both the numerator and denominator by (5 + 2i) gives us [(3 + 4i)(5 + 2i)/(5 - 2i)(5 + 2i)]. This simplifies to (7 + 26i)/29 or after dividing both 7 and 26i by 29,

[28] Why?
[29] Hint: Plot the function!
[30] You may have heard the phrase "imaginary roots" in reference sqrt(-1). This is an unfortunate usage. There is NOTHING imaginary about sqrt(-1) as we'll soon find out!
[31] Remember, there will be the same number of roots as the degree of the function regardless of whether the roots are real or complex.

(0.241379 + 0.896552i).[32] We can also find the distance from any complex ordered pair to the origin (or any other complex ordered pair) by using the Pythagorean Theorem or distance formula. The distance from (4 + 3i) to the origin is simply **`sqrt(`4^2` + `3^2`)`**. We will be using these concepts to create the amazing (literally!) fractals in this and the next two sections.



$x^3 - 1 = 0$

[32] Whew! Don't worry about this overmuch. Ruby handles complex arithmetic automatically!

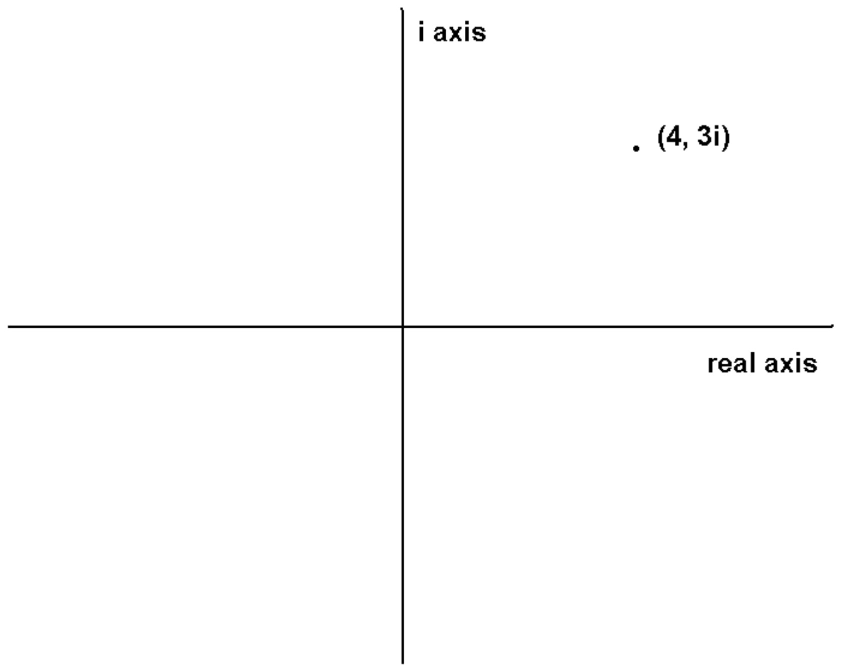


Figure 7.2

So what does all this have to do with Newton's method? Let's see by using a computer program to display the complex roots of $x^2 + 1 = 0$. We are going to substitute the variable z[33] for x in this equation, so we'll be looking at the function $z^2 + 1 = 0$. Here is the listing. Some explanations will be required, so hang in there!

```
# PyNewton.py
# Newton's Method in the complex plane

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

# If psyco isn't installed, delete the next two lines!
import psyco
psyco.full()

# Global variables for screen dimensions, axis range
# and loop step size
width = 400
height = 400
axrng = 2.0
hstep = 2*axrng/width
vstep = 2*axrng/height
```

[33] "z" is a traditional variable used to represent complex numbers.

```
def init():
     # Black background
     glClearColor(0.0, 0.0, 0.0, 0.0)
     gluOrtho2D(-axrng,axrng,-axrng,axrng)

def drawnewton():
     glClear(GL_COLOR_BUFFER_BIT)
     glBegin(GL_POINTS)

     y = axrng
     while y > -axrng:
          y -= vstep
          x = -axrng
          while x < axrng:
               x += hstep

               n = 0

               # define the current complex number
               # using the x,y pixel values
               z = complex(x,y)

               endit = 0

               # 1000 iterations at maximum
               while n < 1000 and endit == 0:
                    n+=1
                    old = z

                    # Newton's Method Equation
                    z = z - (z**2 + 1)/(2*z)

                    if abs(z - old) < 0.000001:
                         endit = 1

               # Pick color parameters based on quadrant
               if z.imag >= 0 and z.real < 1:
                    c1 = 6
                    c2 = 12
                    c3 = 18

               elif z.imag < 0 and z.real < 1:
                    c1 = 18
                    c2 = 6
                    c3 = 12

               if z.real > 0:
                    c1 = 12
                    c2 = 18
                    c3 = 6

               glColor3ub(n*c1,n*c2,n*c3)
```

```
                glVertex2f(x,y)
        glEnd()
        glFlush()

def main():
        glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE)
        glutInitWindowPosition(50, 50)
        glutInitWindowSize(width, height)
        glutInit(sys.argv)
        glutCreateWindow("Newton's Madness")
        glutDisplayFunc(drawnewton)
        init()
        glutMainLoop()

main()

# End Program
```

The Python **Numeric** module also contains complex[34] arithmetical routines. This is absolutely great for our purposes! Now for a short aside… It is my hope that by seeing the fantastic graphics we have been plotting (and will continue to create… the best is yet to be!), that you will become more interested in the mathematics behind the graphics. For this reason, the focus of this course is more on the "pretty pictures" and how to make them rather than on theory. As you continue to study mathematics, perhaps you will be able to fill in the background material and say "So that's how it works!" Anyway, with the **Numeric** module loaded, we can perform complex arithmetic, including complex trig functions without worrying about whether we have the mathematical details of complex functions correct.

The creation of fractal drawings is very calculation intensive, so be prepared to wait for these plots to complete. Sometimes several minutes may pass before the plot is finished. This seems like an eternity while you are watching the screen update line by line, but remember that just a few short years ago, such plots would have taken hours or even days to finish!

The two lines:

```
import psyco
psyco.full()
```

are optional. The **psyco** module, if installed, will speed up Python program execution noticeably in most instances. As stated previously, the creation of fractal drawings is very calculation intensive, so any increase in speed is welcome. This module is free and is available for downloading on the internet. If you don't have the **psyco** module, omit both of these lines from your program.

Next we define some variables for use within the program.

```
width = 400
```

[34] Not "complex" as in complicated, but complex as in **sqrt(-1)**.

```
height = 400
axrng = 2.0
hstep = 2*axrng/width
vstep = 2*axrng/height
```

The window width and height are self-explanatory. We are using a smaller value here (400) in order to speed up the plot somewhat. If you have the patience, you can increase this value to 500 or 600. The beauty of the fractal is more evident at this larger window size, but it takes considerably more time to plot the graphic display! The variable **axrng** allows us to set the viewing window virtual size from **–axrng** to **axrng** in the **gluOrtho2D** statement. In this case, we are using 2.0, which will give us **x** and **y**[35] ranges from -2.00 to 2.00. In this and subsequent programs in this chapter, we need to look at or calculate values for every single pixel in the viewing window. The **hstep** and **vstep** assignment statements automatically do this for us. We calculate the **hstep** and **vstep** values by first doubling **axrng**, which serves to encompass the entire width of the **x** axis and height of the **y** axis. We then divide by the width and height of the window in pixels. The values obtained assure us that we "visit" each pixel in the window. We can then change the window dimensions and **axrng** and be certain that we'll still have the proper pixel step. By the way, if we were to increase the window size to 1000x1000, we would have to perform calculations on 1 million individual pixels!

The **def init():** statement is similar to past programs. You may have wondered what happened to the **def reshape():** function? You may add that subroutine if you like, but remember to add a **glutReshapeFunc(reshape)** to your **def main():** program listing. For now, though, I'm trying to keep the program listings as short and simple as possible due to the complexity of the fractal programs.

In the **def drawnewton():** function, we see a new kind of loop structure.

```
y = axrng
while y > -axrng:
      y -= vstep
      x = -axrng
      while x < axrng:
            x += hstep
```

This is an example of a nested loop similar to the **for** nested loops in the last section.[36] Our goal in this program is to visit every pixel in the display window, so we start **y** at the value for **axrng.** In this example program the initial value for y would be 2.0. The **while y > -axrng:** statement can be interpreted loosely as follows: "as long as y is greater than **–axrng**, keep doing the indented stuff below." We decrease **y** each loop iteration by the **vstep** value (in **y -= vstep**) and then we encounter the **while x < axrng:** loop, which can be interpreted in a similar fashion as the **y** loop. For each single value of the **y** loop, we traverse the entire **x** loop. What this means is

[35] Or "**i**" range, since we are using the complex plane.
[36] So why not use **for** loops? We could, but part of the purpose of this text is to learn some programming skills, hence the use of another looping method. If you want, see if you can rewrite this program using only **for** loops.

that we start with the upper left hand pixel (**`y = axrng`**) and one row of pixels at a time, we cover the entire window. Pay VERY close attention to the indentations in this listing. Again, indentation errors are a major source of problems in Python programs.

Once we begin the loop process, we initialize two "flag" variables and define our complex number seed or initial root guess based on the current x,y pixel value. The pertinent program statements are:

```
n = 0

# define the current complex number
# using the x,y pixel values
z = complex(x,y)

endit = 0
```

We will use the variable **`n`** to count the number of iterations used for Newton's method. The **`endit`** variable, which we'll discuss in more detail in a moment, provides an early escape from the loop if certain conditions are met. The critical line is the **`z = complex(x,y)`** statement. This line takes the current values of x and y (starting with **`x = -2.0`** and **`y = 2.0`**) and converts the ordered pair for each pixel to a complex number. The **`complex(x,y)`** command is provided by the **`Numeric`** module at the beginning of the program.

Newton's method is implemented in the next section:

```
# 1000 iterations at maximum
while n < 1000 and endit == 0:
     n+=1
     old = z

     # Newton's Method Equation
     z = z - (z**2 + 1)/(2*z)

     if abs(z - old) < 0.000001:
          endit = 1
```

Using another **`while`** loop, we arbitrarily decided to allow 1000 iterations as a maximum number[37] and also to provide an early exit from the loop if the variable **`endit`** is anything other than zero. Notice that the loop continues only if BOTH conditions (**`n < 1000`** AND **`endit == 0`**) are true. After the **`while`** statement, we increment the counter variable **`n`** (add 1 to **`n`**) and set the variable **`old`** equal to the last value for **`z`**, which holds the previous root calculation. We need the variable **`old`** so that we can compare the last value for the root of the equation to the new calculated value. The line **`z = z - (z**2 + 1)/(2*z)`** holds the equation (x^2 + 1) and its derivative (2*x) converted to complex expressions involving the variable **`z`** and written according to Newton's formula. The **`z`** variables on the right of the **=** sign hold the initial root seed

[37] 100 iterations is probably more than enough to allow convergence to a root. You can experiment with fewer iterations (or more) if you wish.

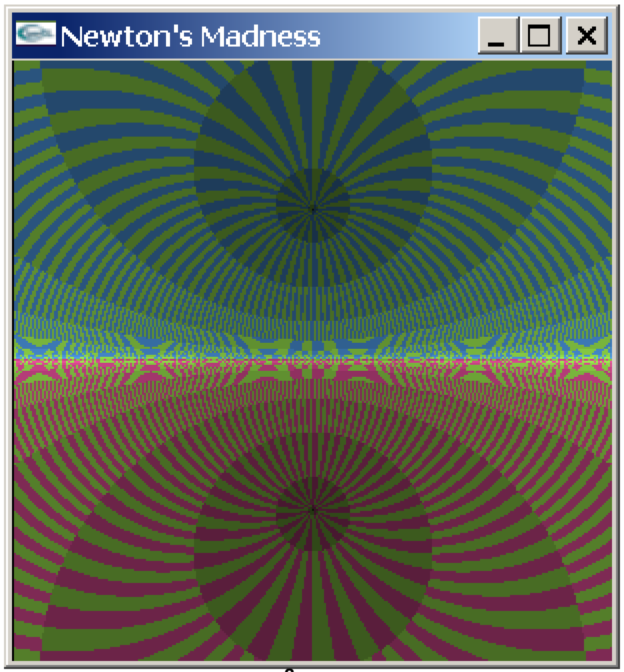
and all subsequent iterations. The calculation based on the previous value of **z** is then stored as the new **z** on the left side of the **=** sign.

Finally, we check to see if the difference between the latest **z** value and the last **z** value (stored in **old**) is small, in this case less than one millionth. If so, then we've probably converged to a root and we can end the iterations for this particular pixel and associated complex ordered pair. When **endit = 1**, the **while** loop stops and we can then plot the result. In other words, all the loop structures take each pixel in the graphics window, convert each pixel's ordered pair to a complex number, and run the resulting complex number through Newton's method to see what happens. We then plot the result in the next section:

```
# Pick color parameters based on quadrant
if z.imag >= 0 and z.real < 1:
     c1 = 6
     c2 = 12
     c3 = 18

elif z.imag < 0 and z.real < 1:
     c1 = 18
     c2 = 6
     c3 = 12

if z.real > 0:
     c1 = 12
     c2 = 18
     c3 = 6

glColor3ub(n*c1,n*c2,n*c3)
glVertex2f(x,y)
```

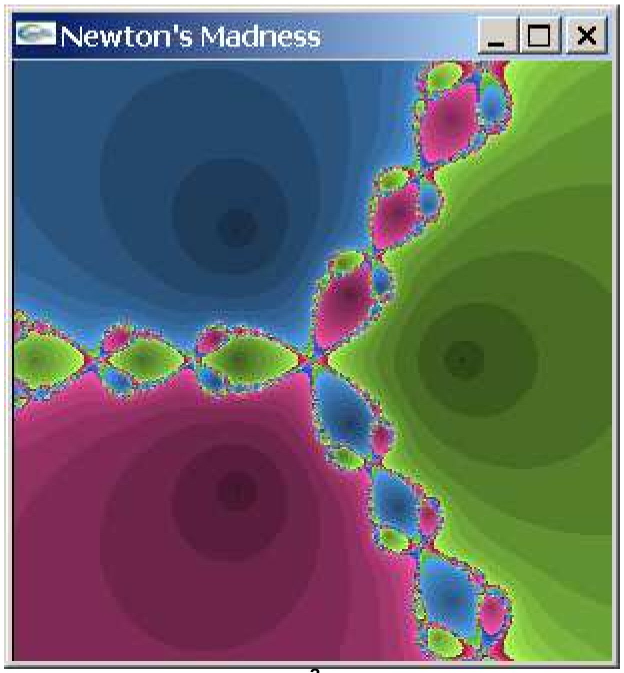
The **z.real** and **z.imag** variables represent the vertical and horizontal complex plane coordinates, analogous to the x and y variables in the standard Cartesian coordinate system. You can think of these variables as x and y values respectively. We are basing the plot coloration on both of the x (**z.real**) and y (**z.imag**) values. The signs of these values determine the quadrant we are in (I, II, III, or IV) and we are basing the coloration on the quadrant. Finally, the **def main():** function is basically the same as in previous programs, reflecting only the changes in function names. Notice the slight difference in the **glColor** statement. The "**ub**" ending stands for "unsigned byte" and is essentially an integer from 0-255 rather than a floating point value from 0-1.0. This means that you can also assign color based on a statement such as **glColor3ub(200, 125, 98)** as well as **glColor3f(0.88, 0.56, 1.0)**.

If everything worked properly, you should see something like figure $\mathbf{z^2 + 1}$ below.



$z^2 + 1$

The two circular basins of attraction on the **y** or complex **i** axis represent the two complex roots for this equation. Now that we know a bit more about Newton's Method and complex numbers, let's return to the more interesting equation $x^3 - 1 = 0$, which is another way of expressing the cube root of one and we will represent as $z^3 - 1 = 0$ in the program. We discussed this function earlier in this section and now we'll see what we can find out about the roots of the equation. Remember that this equation has one real root and two complex roots. To enter this function into our Ruby program, we need the derivative of $z^3 - 1$. Using the power rule in calculus, we find the derivative to be $3z^2$. All we need to do is change the statement `z = z - (z**2 + 1)/(2*z)` in the above example to `z = z - (z**3 - 1)/(3*z**2)` to reflect the new $z^3 - 1$ function and then run the program again. Please be patient! This plot will take several minutes to complete even on a fast computer. Eventually, you should see something like figure $z^3 - 1$ below.



z^3 **– 1**

This is the classic plot generally used to display Newton's method in the complex plane. This plot is amazing in both its complexity and its beauty! Notice the basin of attraction on the x or real axis and the two basins representing the complex roots to the left of the origin and above and below the x axis. Each of the three basins represents a root of the equation **z**^{3} **– 1 = 0.** The roots are found on a circle of radius 1.0 around the origin and are respectively: `z = +1.0`, `z = (-0.5 + i*sqrt(3)/2)`, and `z = (-0.5 – i*sqrt(3)/2)` in clockwise order. The intricate color shades are linked. Any seed, that is, any pixel (representing a complex ordered pair) that is blue converges to the blue basin. Likewise the other colors converge to their respective basins. The complexity of this fractal is infinite. Regardless of how much we magnify the convoluted regions, we never reach the end of the intricate patterns. All three basins of attraction are closely packed together. We are beginning to get a glimpse of the marvelous fractals the complex plane can produce!

In the next section, we'll explore another type of fractal based on an iterative process. This fractal is called the Julia set. But first, let's try some exercises. Pay particular attention to Exercise 5. We'll be using the concept of "zooming" in the next two sections. One final note: Some of these fractal images take a long time to draw. Be patient… I think the wait is worth your time. What takes a few minutes to plot on the computers in our lab would have taken days on the old Apple II's!

Exercises

1) There is a tremendous amount of detail in any fractal drawing. Let's zoom in on the center region of the **z**^{3}**-1** plot. The easiest way to accomplish this would be to change the `axrng` variable. Try `axrng = 0.5` and see what happens. The result

should look something like the figure **`Exercise 1`** at the end of these exercises. We'll explore a more refined zooming technique in Exercise 5.

2) Using different polynomial functions with Newton's method will result in different plots. First, set **`axrng = 2.0`** and then change the **`z = z - (z**2 + 1)/(2*z)`** statement[38] to **`z = z - (z**5 - 1)/(5*z**4)`**. This is the same as finding the roots of the equation **z^5-1**. The result is shown in the figure **`Exercise 2`** below. Note the 5 basins of attraction corresponding to the 5 roots for this equation. How many real and complex roots are there?

3) Using the same equation as in exercise 2, set **`axrng = 0.5`** and run the program again. You should see something like the figure **`Exercise 3`** at the end of this problem set. What happens if you zoom in even further? Try **`axrng = 0.05`**. Now try **`axrng = 0.01`**. Why do you think an error was generated? The smaller the **`axrng`**, the more finely divided is the graph. Could it be that we are reaching the point where z = 0 (or almost) and that causes the overflow error? How could we fix this problem? **NOTE**: Similar to an earlier exercise, the latest version of Python did not generate an error on this exercise.

4) Change the 1000 value in **`while n < 1000 and endit == 0:`** to larger and smaller values. Does the plot change? Also change the 0.000001 in **`if abs(z-old) < 0.000001:`** to larger and smaller values. Again, does the plot change?

5) Coloration is always and important consideration when it comes to graphing or plotting fractals. In this section, we are coding the coloration based on quadrant position. What happens if you comment out the code block:

```
if z.imag >= 0 and z.real < 1:
     c1 = 6
     c2 = 12
     c3 = 18

elif z.imag < 0 and z.real < 1:
     c1 = 18
     c2 = 6
     c3 = 12

if z.real > 0:
     c1 = 12
     c2 = 18
     c3 = 6

glColor3ub(n*c1,n*c2,n*c3)
```

statements and change the **`glColor3ub`** statement to something like **`glColor3f(sin(abs(z)),cos(n),sin(z.real))`**. The results are shown in figure **`Exercise 4`**. Let **`axrng = 0.5`**. Experiment with coloration as you have done in previous exercises.

[38] Or the equivalent "**`z =`**" statement.

6) Until this exercise, we have centered the picture about the origin using a symmetrical window. What happens if you want to view a portion of the graph that is not centered about the origin? We would have to change our **gluOrtho2D** statement and adjust our **vstep** and **hstep** calculations to visit every pixel in the new window. First, let's return the Newton's method program back to its initial state. See the program listing in this section if you forgot to save the original version. Change the variable initializations at the beginning of the program as follows:

```
width = 400
height = 400
hcenter = 0.0
vcenter = 0.0
axrng = 2.0
hstep = 2*axrng/width
vstep = 2*axrng/height
```

In the **def init():** function, change the **gluOrtho2D** statement to:

```
gluOrtho2D(hcenter-axrng, hcenter+axrng, vcenter-axrng, vcenter+axrng)
```

and then modify the **while** loops according to the following in the **def drawnewton():** function:

```
y = vcenter + axrng
while y > vcenter - axrng:
     y-= vstep
     x = hcenter – axrng
     while x < hcenter + axrng:
          x+= hstep
```

These modifications will allow us to choose any point as a "zoom center" and an **axrng** on either side of this center to magnify selected portions of our fractal window. In the example above, we have chosen the origin **(hcenter = 0.0, vcenter = 0.0)** as our new "zoom center" and have defined an **axrng = 2.0** on all sides of the new center. In this particular case, we have created a virtual graphics window with an upper left coordinate of (-2.0, 2.0) and a lower right coordinate of (2.0, -2.0). The **while** loop modifications make certain we visit each point in the corresponding graphics window based on the **axrng** parameter and the new origin we specified with the **hcenter** and **vcenter** coordinates. We also needed to modify the **gluOrtho2D** statement in **def init():** to **gluOrtho2D(hcenter-axrng, hcenter+axrng, vcenter-axrng, vcenter+axrng)** in order for the plot will fill the entire graphics window. If we run this program as is, we'll get the figure **Exercise 5a**, which is exactly the same as the output from figure $z^3 - 1$. This is what we would expect to get! Now let's choose a different zoom center. Let's let **hcenter = 1.0** and **vcenter = 1.0**, keeping **axrng = 2.0**. The resulting plot is in figure **Exercise 5b**. Is this what you would expect? We now have a new "origin" for the graph and it's centered at (1.0, 1.0) with the same +/-2.0 range in both the horizontal and vertical directions.

Now try **`hcenter = 1.1`**, **`vcenter = 1.45`**, and **`axrng = 0.1`**. You should see something like the figure **`Exercise 5c`**. Can you tell where this object is located in the original **`Exercise 5a`** plot? So, in conclusion, the use of the global variables **`hcenter`** and **`vcenter`**, coupled with the axis range variable **`axrng`** will literally allow us to move around and explore various regions of a fractal image AND zoom into or away from each region. Make certain you understand this concept before you go further!

7) We can also "zoom out" for a larger perspective. Set **`hcenter = 0.0`** and **`vcenter = 0.0`** and then let **`axrng = 6.0`** and see what happens. The figure **`Exercise 6`** illustrates the "zoom out" feature.

8) The following is a list of new equations to try. I recommend that you return the program to its original state as presented in the first part of this section, but include the zoom modifications we made in Exercise 5 (**`hcenter`** and **`vcenter`**). The expression under the "**`/`**" sign is the derivative of the expression in the numerator. You can try zooming and/or changing the origin of the graph. Experiment! Although it probably won't be Newton's method, you may also make up your own equation expressions. Remember that most of these fractals take a long time to plot, so be patient. If you find that the program generates an overflow or math range error, see if you can figure out what is causing the error and either fix or trap the error so it does not occur. Figures representing each equation can be found at the end of the exercises and labeled **`Exercise 7a`** through **`Exercise 7f`**. Hint: You can enter several equations into your program at the same time. Simply comment out all of the equations **except** the one you want to plot.

```
# Newton's Method Equations
a) z = z - sin(z)/cos(z)
b) z = z - (z**3 + z**2 + z - 2)/(3*z**2 + 2*z + 1)
c) z = z - (z**4 + .84*z**3 + .16*z - 2)/(4*z**3 + 2.52*z**2 + .16)
d) z = z - (z**4 + .84*z**2 - .16)/(4*z**3 + 1.68*z)
e) z = z - log(z)/(1/z)
f) z = z - (z**3 - 5*z)/(3*z**2 - 5)
```

DON'T type the a, b, c, d, etc. Those letters are there to help you find the appropriate figures after the exercises!

If you find that errors are generated during a plot and you can't fix or trap them, you can have Python ignore the error completely! Here's how:

```
try:
   z = z – log(z)/(1/z)
   # other equations here
   # and here
except:
   pass
```

Essentially, this code block says "**`try`** the following statement and if an error occurs, do whatever is after the **`except`** statement". In this case, we simply pass to the next line of code. So, if you place **`try:`** above the equation set (indenting the equations!)

and **`except:`** after the equation set (with an indented **`pass`** below **`except:`**) we should be able to use any of these equations without the fear of generating an error.

9) We are going to take a closer look at the function **`z = z - sin(z)/cos(z)`** from exercise 8. The figure that represents this plot is found in figure **`Exercise 7a`**. Uncomment the equation for this function (commenting all others!) into your Newton's method program and modify **`hcenter`** to **`hcenter = 1.55`** and **`axrng`** to **`axrng = 0.50`**. These modifications should center and zoom into the interesting figure on the right side of the graphics window. Your plot should look something like the plot shown in figure **`Exercise 9a`**. Now set **`axrng = 0.30`** and run the program again. You should see something like figure **`Exercise 9b`**.

10) We don't have to use integer powers when using Newton's Method. We can find the complex iteration of $z^{3.5} - 1$ just as easily as we can an integer power. Modify the equation line as follows (or simply add this equation to your list):

```
   z = z - (z**3.5 - 1)/(3.5*z**2.5)
```

Using the color code block from the original Newton's Method program, we will get a plot similar to the one shown in figure **`Exercise 10a`**. A somewhat more interesting plot can be found by comment out the original color code block (as we did in exercise 4) and substituting this color assignment prior to the **`glVertex2f`** statement:

```
   if z.imag < 0:
         glColor3f(sin(n),cos(n),sin(z.real))
   else:
         glColor3ub(5*abs(z),cos(z.real),1.5*abs(z))
```

The resulting plot is found in figure **`Exercise 10b`**. I didn't say it was pretty!

You can also try other z equations (with any color scheme) such as:

```
   z = z - (z**4.5 - 1)/(4.5*z**3.5)
   z = z – (z**4.8 – 1)/(4.8*z**3.8)
   z = z – (z**3.75 – 1)/(3.75*z**2.75)
```

By now you may have some idea of how to find the derivative of a simple polynomial function? If so, you might try making up some of your own **`z`** equations.

11) Here is a "small" list of equations you may want to try in your program. Make certain that you comment out every equation except the one you want to try!

```
# Newton's Method Equation
#z = z - (sin(z)/cos(z) - 1)/(1/((cos(z)*cos(z))))
#z = z - (z**4 + .84*z**2 - .16)/(4*z**3 + 1.68*z)
#z = z - (z**3 - 1)/(3*z**2)
#z = z - sin(z)/cos(z)
z = z - (z**5 - 1)/(5*z**4)
#z = z - log(z)/(1/z)
#z = z - (z**3 + z**2 + z - 2)/(3*z**2 + 2*z + 1)
#z = z - (z**4 + .84*z**3 + .16*z - 2)/(4*z**3 + 2.52*z**2 + .16)
#z = z - (z**4 + .84*z**2 - .16)/(4*z**3 + 1.68*z)
```

```
#z = z - (z**3 - 5*z)/(3*z**2 - 5)
#z = z - (sin(z*z) - z*z + 1)/(2*z*cos(z*z) - 2*z)
#z = z - (z**3.7 - 1)/(3.7*z**2.7)
#z = z - (sin(z*z) - z*z*z + cos(z*z*z))/(2*z*cos(z*z) - 3*z*z - 3*z*z*sin(z*z*z))
#z = z - (z**z - 3**z - z**3 - 1)/((1+log(z))*z**z - log(3)*3**z - 3*z**2)
```

Can you tell which equation we are trying to plot?

One final modification you might try is to uncomment TWO equations at the same time. What will happen? I don't know... try it! Here's what I THINK will happen. **`z`** will be calculated using the first equation and then the complex value for **`z`** will be "fed" to the second equation for calculation/iteration. Certainly the plot will take longer to create. What is uncertain is how interesting the final result will be. This is an excellent opportunity for you to experiment!

12) There are some excellent freeware fractal programs available online. I would recommend that you Google for “fractals” and “freeware” and see what you can find. Some of the current programs that you might find very interesting are “Xaos”, “Chaospro”, and “Fractint”. There may be others as well. What we are programming here can hardly be termed a professional application, but hopefully you will get the flavor of fractals and find them interesting enough to explore them further on your own. Another interesting program is Gnofract 4D, which runs on linux. You also should definitely look up "Newton's Method Fractals" and see what you find. You may be able to take the information you find online and convert the equations to a Python program. Give yourself extra credit if you can do so!

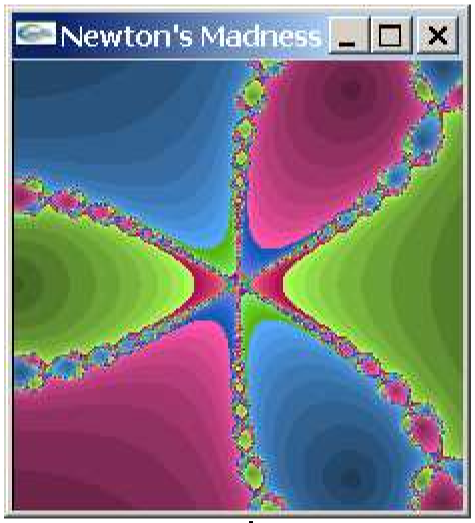
13) One of the most important aspects of a computer program is interactivity. In a future section we will explore mouse interaction with our graphics window. In this exercise we are going to add some additional keyboard options. Until now, in order to view a different Newton's Method fractal we had to comment and uncomment specific equations. What if we could simply press a number key and have the equations automatically change for us? Here's how we might do that. First, add a new global variable, **`global newtfrac`**, to the variable section at the beginning of the program and set its initial value equal to 1 (How?). Add the following **`def keyboard`** function to your program:

```
def keyboard(key, x, y):
     global newtfrac
     if key == chr(27) or key == "q":
          sys.exit()
     else:
          newtfrac = eval(key)
          if newtfrac > 0 and newtfrac < 10:
               glutPostRedisplay()
```

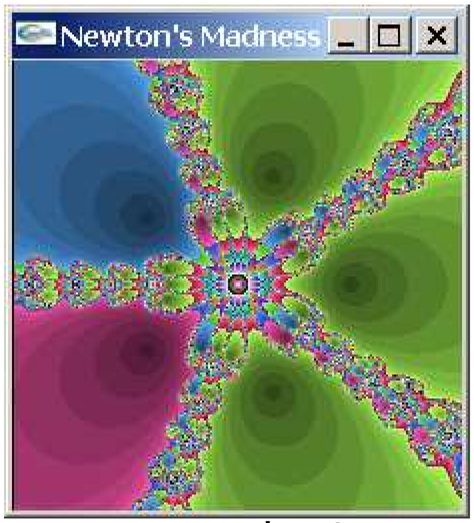
Since we are going to change the value of the **`newtfrac`** variable, we need to declare it as **`global`** in this **`def keyboard`** function. The program will end if we press the "q" or "Esc" keys. If we press any other key, its numeric value will be stored in **`newtfrac`** by the **`eval(key)`** function. If this value is between 0 and 10 (in other words, 1 through 9), then we update the display. Note: What statement must we add to **`def main()`** in order for the **`keyboard`** function to work? The real work is done in the display function. Modify your display function by changing/adding the following:

```
while n < 1000 and endit == 0:
     n+=1
     old = z

     # Newton's Method Equation
     try:
          if newtfrac == 1:
               z = z - (z**3 - 1)/(3*z**2)
          elif newtfrac == 2:
               z = z - (z**4 + .84*z**2)/(4*z**3 + 1.68*z)
          elif newtfrac == 3:
               z = z - (z**5.4-1)/(5.4*z**4.4)
          elif newtfrac == 4:
               z = z - sin(z)/cos(z)
          elif newtfrac == 5:
               z = z - (z**5 - 1)/(5*z**4)
          elif newtfrac == 6:
               z = z - log(z)/(1/z)
          elif newtfrac == 7:
               z = z - (z**3.7 - 1)/(3.7*z**2.7)
          elif newtfrac == 8:
               z = z - (3**z - 1)/(log(3)*3**z)
          else:
               z = z - (z**3 - 5*z)/(3*z**2 - 5)
     except:
          pass

     if abs(z - old) < 0.000001:
          endit = 1
```

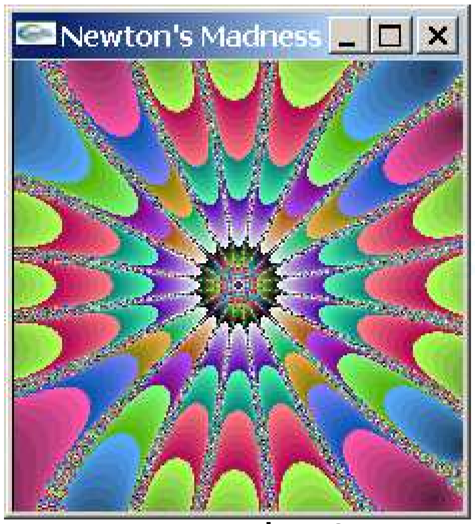
The above is not completely new, obviously. Simply modify the appropriate section of code in the **`def drawnewton()`** function to match what you see here. Run the program. Once the original Newton's Method fractal has finished drawing, press any number key other than 1 or 0 and see what happens. How does this work? Look carefully at the **`if...elif...else`** code block and see if you can understand the logic. How does this particular block of code work with the keyboard function? Why did I not need to specifically check for the number "9"? If 9 works, why doesn't the number "0" trigger the **`else`** statement?



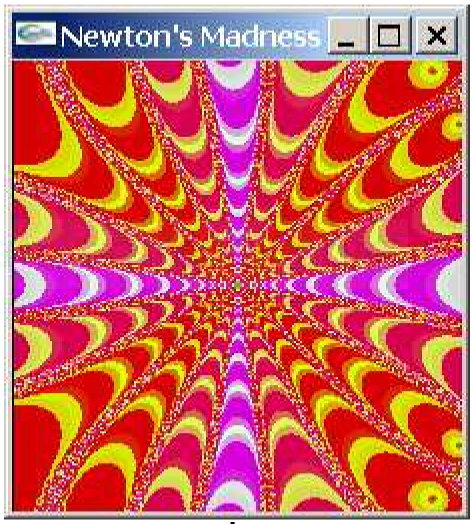
Exercise 1



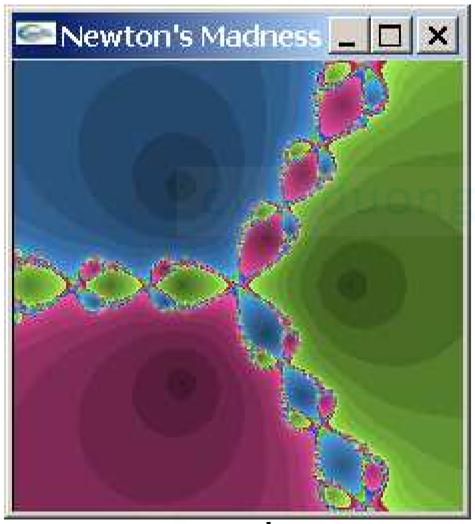
Exercise 2



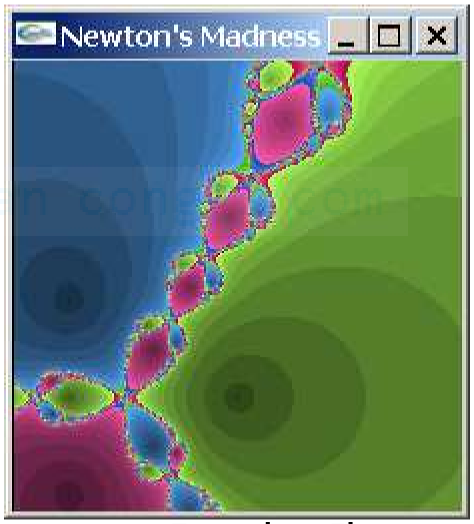
Exercise 3



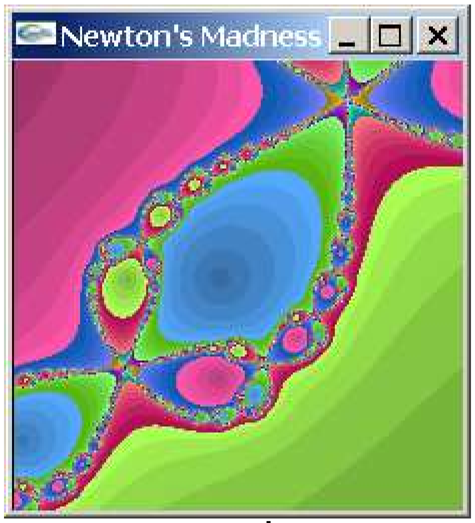
Exercise 4



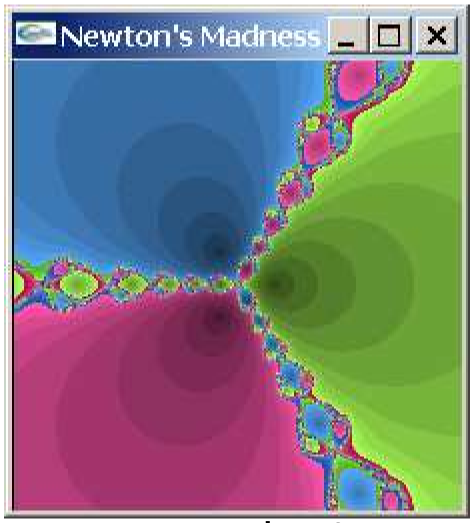
Exercise 5a



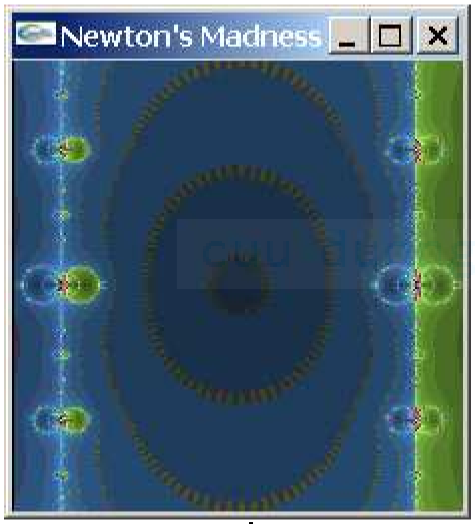
Exercise 5b



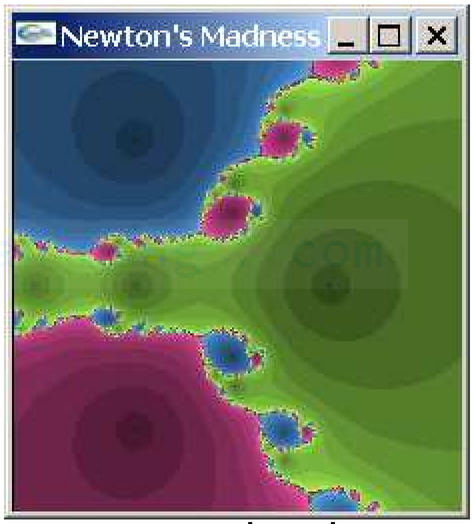
Exercise 5c



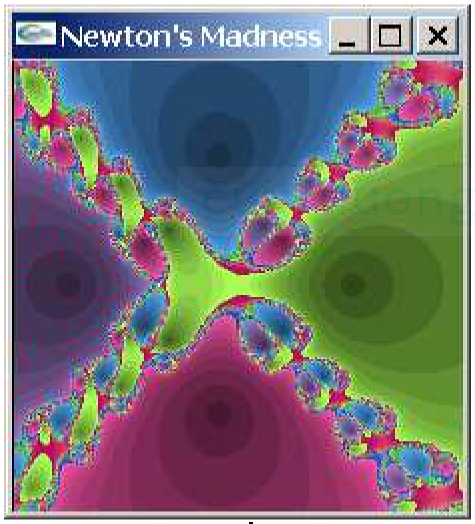
Exercise 6



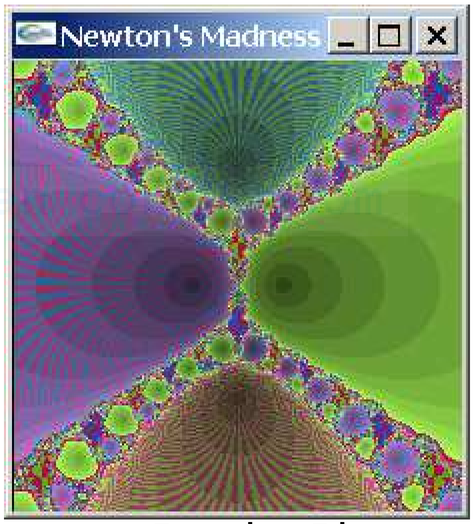
Exercise 7a



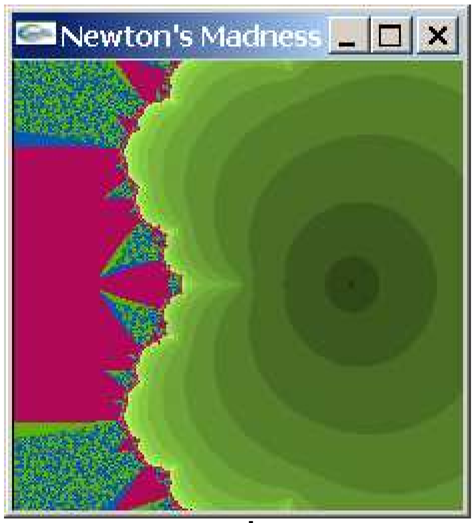
Exercise 7b



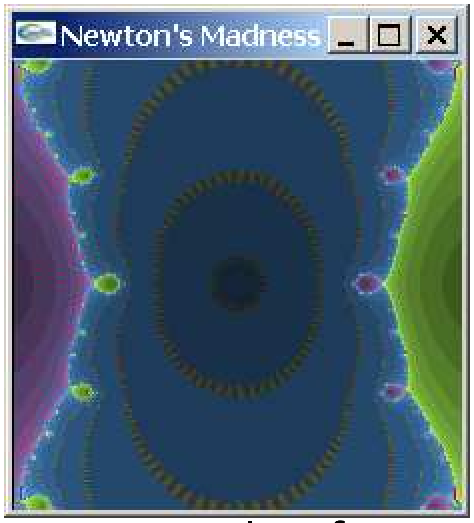
Exercise 7c



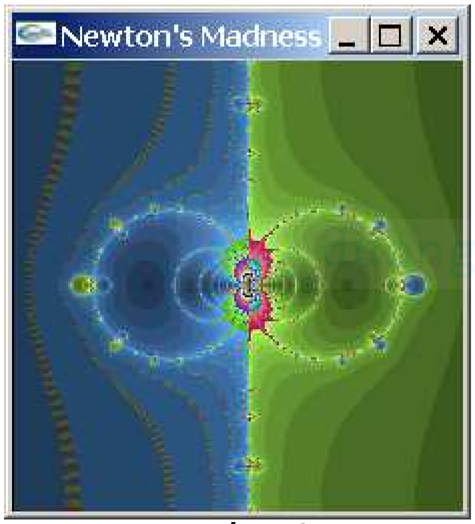
Exercise 7d



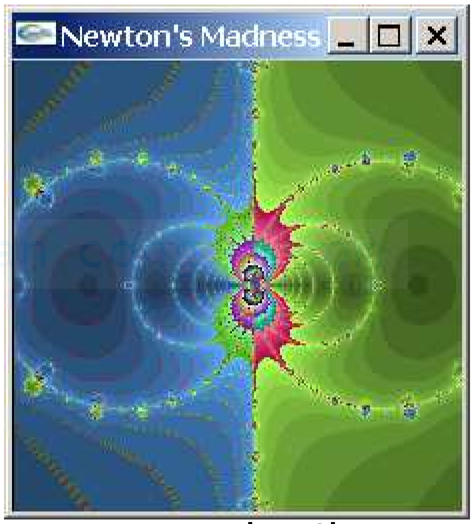
Exercise 7e



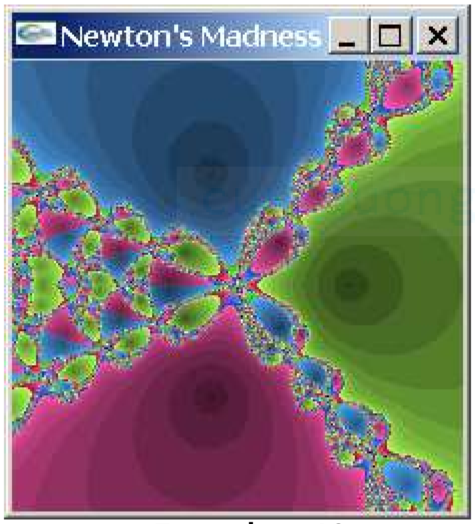
Exercise 7f



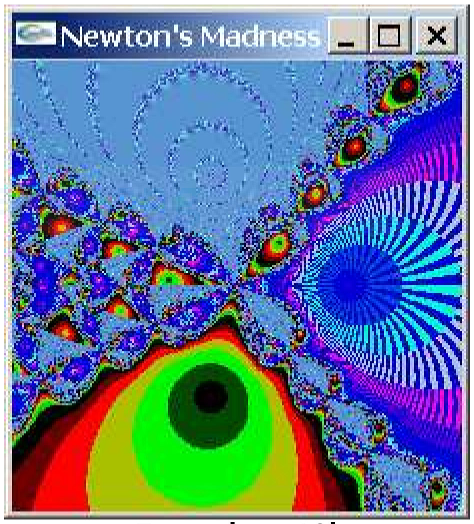
Exercise 9a



Exercise 9b



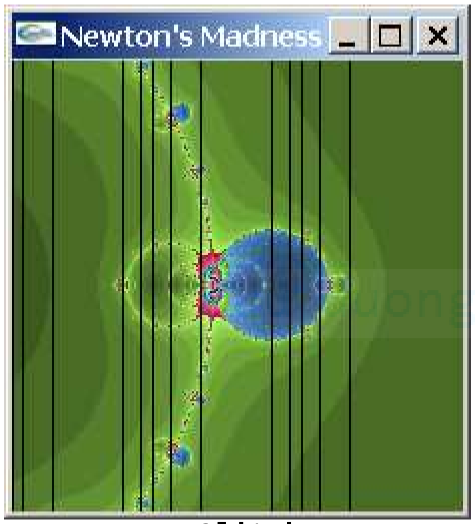
Exercise 10a



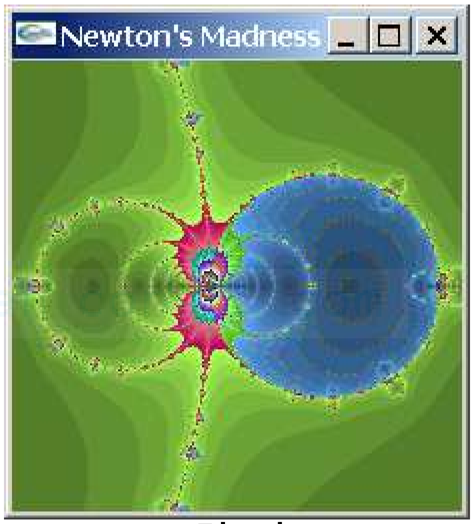
Exercise 10b

Addendum:

I want to encourage you to explore some of these (and other) fractals on your own. Try zooming in on various locations and see what happens. If you generate an error message, what might be causing the error? Are you dividing by zero? Figure **`glitch`** on the next page illustrates another type of error. In this figure, **`hcenter = 1.925`** and **`axrng = 0.10`** using the same equation as in Exercise 8. What caused the dark vertical lines to appear? How might you fix this problem? Figure **`Fixed`** used **`hcenter = 1.92`** and **`axrng = 0.05`**. Hint: Add 1 to each denominator in the **`hstep`** and **`vstep`** global variable assignments, i.e. **`(width+1)`**.



`Glitch`



`Fixed`

Addendum II:

One final modification you can attempt. In your ORIGINAL **pynewton.py** program, change the:

```
z = complex(x,y)
```

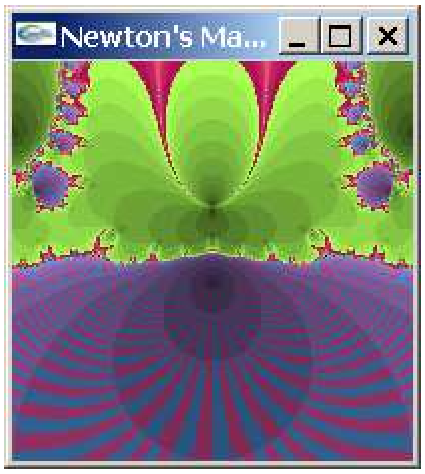
statement to:

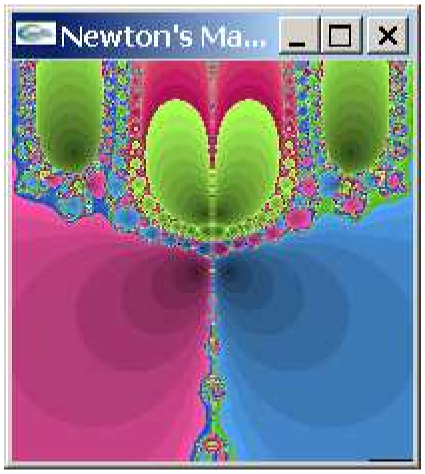
```
z = complex(y,x)  #rotates the plot 90 degrees c-clockwise
```

and then try the following two equations ONE at a time:

```
z = z - (z*z - 2**z - 1)/(2*z - 2**z*log(2))
z = z - (z**3 - 3**z - 1)/(3*z**2 - 3**z*log(3))
```

In order, the 2 figures below illustrate the respective results. Stunning!


Newton's Ma...


Newton's Ma...

Section 7.6 The Julia Set

What would happen if you take a function such as $x = x^2 + 1$ and iterate the function, starting at $x = 0$? You would get the following results in the first 6 iterations:

$x = (0)^2 + 1 \rightarrow x = 1$
$x = (1)^2 + 1 \rightarrow x = 2$
$x = (2)^2 + 1 \rightarrow x = 5$
$x = (5)^2 + 1 \rightarrow x = 26$
$x = (26)^2 + 1 \rightarrow x = 677$
$x = (677)^2 + 1 \rightarrow x = 458330$

As you can see, the value for x quickly becomes very large and will grow toward infinity as the iterations continue. Devaney[39] defined the sequence of values calculated from iterating a function the "orbit" of that function. Now let's change the function to $x = x^2 – 1$ and see what happens to the orbit over 6 iterations.

$x = (0)^2 - 1 \rightarrow x = -1$
$x = (-1)^2 - 1 \rightarrow x = 0$
$x = (0)^2 - 1 \rightarrow x = -1$
$x = (-1)^2 - 1 \rightarrow x = 0$
$x = (0)^2 - 1 \rightarrow x = -1$
$x = (-1)^2 - 1 \rightarrow x = 0$

This is an example of a nicely behaved periodic oscillation. We will always get these same two values no matter how long we iterate the function. You would probably guess that if we use a function such as $\mathbf{x = x^2 + 0.5}$, we'll eventually see the values for x "run away" toward infinity and you would be correct! But what about $\mathbf{x = x^2 – 0.5}$? The result is not at all obvious. Here are the first 6 iterations using a hand-held calculator:

$x = (0)^2 - 0.5 \rightarrow x = -0.5$
$x = (-0.5)^2 – 0.5 \rightarrow x = -0.25$
$x = (-0.25)^2 – 0.5 \rightarrow x = -0.4375$
$x = (-0.4375)^2 – 0.5 \rightarrow x = -0.30859375$
$x = (-0.30859375)^2 – 0.5 \rightarrow x = -0.404769897$
$x = (-0.404769897)^2 – 0.5 \rightarrow x = -0.33516133$

After many iterations[40] the value settles down to -0.3660254037844386 and oscillates back and forth between a 6 and 7 in the final decimal place. Finally, what about the function $\mathbf{x = x^2 – 1.57}$? When iterating this equation 1 million times, the orbit never seems to settle down at all. The final 5 iterations are:

<u>Iter.</u>	<u>x-value</u>
999996	0.0023399937526939 38
999997	-1.569994576881325

[39] Devaney, R. L. (1992). "A First Course in Chaotic Dynamical Systems". Perseus Books: Cambridge, MA. Pages 17-32.
[40] 1000 iterations using a short Python program. Can you write a program to check this?

```
999998      0.8948829189846828
999999      -0.7691846137615411
1000000     -0.9783550824045962
```

What would be the value for x after the 1000001st iteration? I don't know without checking the results of my Python program. You could even write a program to check my work if you wish, but chances are you would not get the same results I have listed unless you use exactly the same precision arithmetic I used[41]… remember chaos? Very tiny differences in precision between our programs (or perhaps even our computers?) would feedback and the errors would grow until our two orbits would not even be close to each other.

So, what's the point of this lesson? It appears that iterated functions can do one of 3 things. They can head toward infinity (either positive or negative), they can find a steady state (a single value such as zero, or a sequence of repeating values), or they can appear to behave randomly, dare I say chaotically, without any pattern at all. Things get even more interesting if we consider complex functions. What would happen if we were to choose a complex number such as (**-0.74543, 0.11301**i) and iterate this number using at every point in the complex plane while using the same unique function? We could set some limitations on the function such that if the function's modulus (distance from the calculated complex point and the origin) goes beyond a certain distance, then we plot the point using a particular color scheme. If the point does not stray far from the origin, then we either leave the point alone (don't plot the point) or we use a different color scheme. The resulting plot is called a Julia Set in honor of Gaston Julia[42], a mathematician who worked with such sets prior to the advent of the computer. Julia could only imagine what incredible "monsters" he discovered! It took the invention of the computer for us to actually visual these sets and their incredible beauty.

To plot a Julia Set for the complex number (**-0.74543, 0.11301**i), use the following program listing:

```
# PyJulia.py
# Plot a Julia set

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

# If psyco isn't installed, delete the next two lines!
import psyco
psyco.full()

# Initalize screen dimensions and the screen origin
width = 400
```

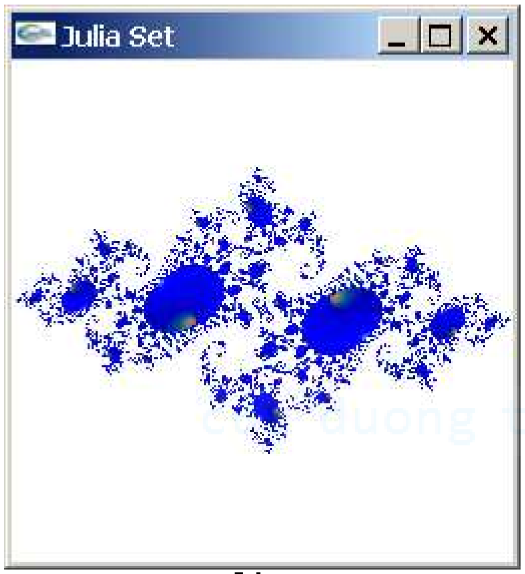
[41] I wrote a very short QBasic program using double-precision arithmetic to calculate the orbit of this function. If you didn't use double-precision arithmetic and/or QBasic, you would NOT get the same results I did. Chaos in action!
[42] http://www.fractovia.org/people/julia.html

```
height = 400
hcenter = 0.0
vcenter = 0.0
axrng = 1.5
hstep = 2*axrng/width
vstep = 2*axrng/height

def init():
     # White background
     glClearColor(1.0, 1.0, 1.0, 1.0)

     # The next statement is all one line!!!
     gluOrtho2D(hcenter-axrng,hcenter+axrng,vcenter-
     axrng,vcenter+axrng)

def drawjulia():
     glClear(GL_COLOR_BUFFER_BIT)
     glBegin(GL_POINTS)

     # Julia set complex number
     z = complex(-0.74543, 0.11301)

     y = vcenter + axrng
     while y > vcenter - axrng:
        y-= vstep
        x = hcenter - axrng
        while x < hcenter + axrng:
         x+= hstep

         n = 0
         a = complex(x,y)

         # n < 100 is the number of iterations
         # Increase this value to show finer detail
         # Decrease the value if nothing shows on the screen
         while n < 100:
            n+=1
            a = a**2 + z
            zz = abs(a)

            # zz > 2 is the critical escape value
            # Some functions require larger escape values
            # This zz > 2 conditional provides coloration for
            # points outside the Julia set
            if zz > 2:
               #glColor3f(sin(2*zz),cos(zz),sin(4*zz))
               #glVertex2f(x,y)
               n = 5001

         # This zz < 2 conditional provides coloration for
         # points inside the Julia set.
         if zz < 2:
                glColor3f(cos(5*zz),sin(4*zz)*cos(4*zz),sin(2*zz))
                glVertex2f(x,y)
        glEnd()
        glFlush()

def main():
        glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE)
        glutInitWindowPosition(50, 50)
        glutInitWindowSize(width, height)
        glutInit(sys.argv)
        glutCreateWindow("Julia Set")
        glutDisplayFunc(drawjulia)
        init()
        glutMainLoop()

main()

# End Program
```

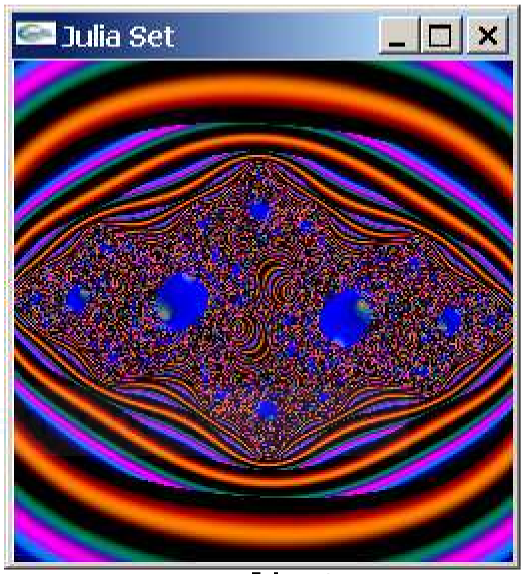
The Julia Set plot from the above listing is shown in figure **`Julia 1`** below. If we uncomment the lines below the **`if zz > 2:`**

```
            #glColor3f(sin(2*zz),cos(zz),(sin(4*zz))
            #glVertex2f(x,y)
```

In the **`def drawjulia():`** function we'll get something similar to figure **`Julia 2`**.



Julia 1



Julia 2

The code for the Julia Set does not differ greatly from the code for Newton's method in the last section. The major difference is found in the display routine:

```
def drawjulia():
        glClear(GL_COLOR_BUFFER_BIT)
        glBegin(GL_POINTS)

        # Julia set complex number
        z = complex(-0.74543, 0.11301)

        y = vcenter + axrng
        while y > vcenter - axrng:
           y-= vstep
           x = hcenter - axrng
           while x < hcenter + axrng:
             x+= hstep

             n = 0
             a = complex(x,y)

             # n < 100 is the number of iterations
             # Increase this value to show finer detail
             # Decrease the value if nothing shows on the screen
             while n < 100:
                n+=1
                a = a**2 + z
                zz = abs(a)

                # zz > 2 is the critical escape value
                # Some functions require larger escape values
                # This zz > 2 conditional provides coloration for
                # points outside the Julia set
                if zz > 2:
                   #glColor3f(sin(2*zz),cos(zz),(sin(4*zz))
                   #glVertex2f(x,y)
                   n = 5001

             # This zz < 2 conditional provides coloration for
             # points inside the Julia set.
             if zz < 2:
                glColor3f(cos(5*zz),sin(4*zz)*cos(4*zz),sin(2*zz))
                glVertex2f(x,y)
        glEnd()
        glFlush()

        # End Function
```

Notice that we are choosing a complex number in the

z = complex(-0.74543, 0.11301)

line near the beginning of the **def drawjulia():** function. If we change the complex number, we'll change the appearance of the Julia Set… already we have something we can experiment with! We then choose every single pixel in the graphics window and change each pixel to a complex number **`a`** within the nested **`while`** loop structure using:

```
a = complex(x,y)
```

The Julia Set is iterated in the **`while n < 100:`** loop using the **`a = a**2 + z`** statement. The variable **`z`** contains the original Julia Set seed and the variable **`a`** represents the complex number corresponding to each pixel in the graphics window. We iterate EACH pixels complex number representation until a predetermined number of iterations is reached OR the value of the complex function exceeds a certain number or modulus. It is known that if the orbit of the Julia Set ever exceeds a modulus[43] of 2, then the orbit will escape to infinity. That is the purpose of the lines:

```
zz = abs(a)
if zz > 2:
```

If the orbit distance or modulus, which we represent with the **`abs(a)`** function, exceeds 2, then we want to stop the iteration process and either do nothing or plot the point, subject to the coloration functions in the **`glColor3f`** statement. Such points that "run away to infinity" are NOT in the Julia Set. If we iterate the function 100 times and the modulus does NOT exceed 2, then we know (or we think we know[44]) that this particular complex number (pixel) is IN the Julia Set and we plot that point. The end result of iterating all pixels in the window subject to the original complex number (**`-0.74543, 0.11301`**i) is the Julia Set. I think you will agree that Julia Sets are striking in their appearance!

We'll explore Julia Sets and the Julia Set code more thoroughly in the Exercises. In the next section, we'll end our formal 2D non-animation instruction with the grandfather of all fractals, the Mandelbrot Set. The Mandelbrot Set serves as a catalog of all possible Julia Sets and has been termed the most complicated mathematical object ever discovered. Whether this is true or not is probably open to interpretation, but the Mandelbrot Set is unforgettable in its appearance and infinitely detailed.[45] As mentioned in exercise 9 in the previous section, I strongly recommend that you obtain a fractal freeware program to explore and create your own fractals and fractal types.

Exercises

1) Among the more obvious modifications to the Julia Set program is to change the initial complex number. You can experiment with this on your own. Just to get you started, try **`z = complex(-0.5, 0.6)`**. I also recommend that you change the

[43] Remember the modulus is the distance from the complex point to the origin. It is found by using the Pythagorean theorem. Python takes care of the calculation for you in the **`abs(a)`** function, where **`a`** is a complex number.
[44] 100 iterations is NOT enough to make such decisions. In order to refine the Julia Set, we need to iterate much more than this number. 500, 1000, 10000 iterations would be better, but we don't have that much time, do we? Actually, an infinite number of iterations would be required to decide whether some points on the border of the Julia Set actually belonged to the set.
[45] As are most (all?) fractals, including the ones generated by Newton's Method and the Julia Set.

background color to black using the **`glClearColor`** statement in **`def init():`**. The plot from the complex number in this exercise is shown in figure **`Exercise 1`** at the end of these exercises. Keep the range of values for both the real and the complex numbers between -1.0 and +1.0.

2) You can create a more detailed plot by increasing the number of iterations in the **`while n < 100:`** loop. Change the 100 to 500 and see what happens. Using the complex number from exercise 1, the plot with 500 possible iterations is shown in figure **`Exercise 2`**. There isn't much difference at this zoom level, but we'll explore this concept more in exercise 3. What else do you notice about the creation of this new 500 maximum iteration Julia Set? Obtaining more detail in a drawing isn't free! If you want, you can increase the **`width`** and **`height`** of the graphic window, but again, a larger Julia Set plot isn't free.

3) Let's zoom into a portion of the Julia Set we created in exercise 1 and illustrate the concept in exercise 2 a bit more explicitly. First, set the number of iterations to a low number by changing the iteration loop to **`while n < 10:`**. Also, set **`hcenter = 1.0`**, **`vcenter = -0.5`**, and **`axrng = 0.1`**. The resulting plot is shown in figure **`Exercise 3:  n<10`**. Not very appealing, is it? Now change the number of iterations to 50. The difference in plots is huge and is demonstrated in figure **`Exercise 3:  n<50`**. Now change the number of iterations to 1000 (be patient!). The result is shown in figure **`Exercise 3:  n<1000`**. When creating fractal images there is usually a trade-off between speed and the viewing of details. Notice in the upper right hand portion of the **`Exercise 3: n<50`** and **`Exercise 3: n<1000`** plots. You can easily see that that **`n<1000`** plot displays a finer level of detail, but you have to look closely to notice the difference from the **`n<50`** plot. Generally, the larger the plot window, the more detailed your plot should be. Remember, however, that the plot time will be longer. If you notice some dark vertical line artifacts, remember how we fixed this in the last section by adding 1 to each denominator in the **`hstep`** and **`vstep`** statements at the beginning of the program.

   There is a limit to how precisely we can draw a fractal. While fractals themselves are infinitely defined, your computer screen is not. Details of any fractal that are less than the size of a single pixel can't be displayed. This is why we must seek a compromise between the number of iterations (which determines the fineness of fractal details) and speed of drawing. Try **`while n<5000:`** and see if you can tell the difference between 5000 iterations and 1000 iterations on your monitor. The result of 5000 iterations is shown in figure **`Exercise 3: n<5000`** at the end of these exercises.

4) Adding extra color to fractal plots usually creates some interesting effects. Usually we color fractals such as Julia and Mandelbrot sets by escape time, that is, by how many iterations it takes for the complex point being examined to escape to infinity. We can also color a fractal based on the distance the iterated point is from the origin when the maximum number of iterations is reached. Look at the following Julia set code again (# comment statements omitted):

```
while n < 100:
```

```
            n+=1
            a = a**2 + z
            zz = abs(a)
            if zz > 2:
               #glColor3f(sin(2*zz),cos(zz),(sin(4*zz))
               #glVertex2f(x,y)
               n = 5001

         if zz < 2:
            glColor3f(cos(5*zz),sin(4*zz)*cos(4*zz),sin(2*zz))
            glVertex2f(x,y)
```

Uncomment the two lines under the **`if zz > 2:`** statement and change the **`glColor3f`** statement under the **`if zz < 2:`** to the following:

```
glColor3f(sin(n)*zz,sin(n)*zz,cos(n)*tan(zz))
```

The coloration is now based on **`n`**, the number of iterations, and **`zz`**, the distance of the iterated point from the origin. The first **`glColor3f`** statement colors the points that escape to infinity[46] and the second **`glColor3f`** statement colors the points that are inside the Julia Set.[47] The result of these changes is shown in figure **`Exercise 4`** below this exercise set. Experiment with these color statements and see what you can create. Also, remember that you can use various **`axrng`** (zoom) values as well as changing the initial complex number and the **`hcenter`** and **`vcenter`** parameters. The number of possible Julia Set fractals is literally infinite!

5) In this exercise, we are going to explore a few interesting initial complex numbers that lead to some classic Julia Sets. First, return the initial parameters back to their starting values:

```
hcenter = 0.0
vcenter = 0.0
axrng = 1.5
```

which will return the screen center to the origin and set the axis ranges to +/- 1.5. Also, return the remainder of the program back to its original state, including commenting out the **`glColor3f`** and **`glVertex2f`** statements we uncommented in the last exercise. Keep the background color black by using **`glClearColor(0.0, 0.0, 0.0, 1.0)`** in the **`def init():`** function. Also use the original Julia Set complex number **`z = complex(-.75,.1)`**. Now change the **`glColor3f`** statement under the **`if zz < 2:`** conditional to:

```
glColor3f(tan(zz),zz*sin(zz),tan(zz))
```

[46] As stated earlier, we stop iteration if the distance from the origin to the point is greater than 2. Once the orbit of a Julia or Mandelbrot set exceeds 2, then the orbit always goes to infinity.
[47] The Julia Set is actually the boundary between the interior points and the points that escape to infinity.

The resulting plot can be viewed in figure **`Exercise 5a`**. Now try the following initial complex Julia Set seeds (each plot can be viewed in the corresponding **`Exercise 5 (x,yi)`** figures following the exercises:

```
z = complex(-1, 0.0)
z = complex(-0.4, -0.6)
z = complex(-1.5, 0.0)
z = complex(-0.8, 0.0)
z = complex(-0.1, 0.8)
z = complex(0.3, -0.4)
z = complex(-0.5, 0.57)
z = complex(-0.11, 0.86)
z = complex(0.28, 0.53)
z = complex(-0.1, 0.75)
```

What happens to the plots if you switch or change the signs of these complex numbers? Do you see any patterns based on the initial complex coordinates? If you try your own initial points and nothing appears, change the **`glColor3f`** statement or uncomment the **`glColor2f`** statement and its corresponding **`glVertex2f`** under the **`if zz > 2:`** statement. You might research the **`glColor3ub`** statement and see if you can use this command for coloration. The "**`ub`**" ending, as stated earlier, stands for "unsigned byte" and the parameters within the **`glColor3ub`** statement should be integers from 0-255. Coloration is critical in these fractals!

6) The main Julia Set engine for generating the unique fractal images displayed thus far is found in the line:

```
a = a**2 + z
```

within the **`while n < 100:`** iteration loop. We are not limited to this expression and can find the Julia Set of many different equation forms if we wish. To begin this exercise, set **`z = complex(-0.75, 0.05)`** and change the second **`glColor3f`** statement to **`glColor3f(tan(zz)*zz,cos(zz)*zz,tan(zz)*zz)`**. With a white background color, the resulting plot should resemble figure **`Exercise 6a`** below. Now change the Julia Set equation to:

```
a = sin(a) + z
```

and run the program again. Be patient… trig functions take time. You should get a plot that resembles the blobby structure in figure **`Exercise 6b`**. Not very interesting, is it? OK, let's experiment. First change the escape time values found in the **`if zz`** statements from 2 to 50 in each. Run the plot again (be patient!). You should see something like figure **`Exercise 6c`**. Hmmm… there's some structure, but not much else. OK, let's zoom out. Set **`axrng = 5.0`**. Figure **`Exercise 6d`** illustrates the result. That's a bit better, but the colors are terrible! I'll let you adjust the coloration… see what you can do to make this plot more interesting. You should also attempt to use **`a = cos(a) + z`**. What difference, if any, do you see between the **`sin(a)`** and **`cos(a)`** plots. Look closely! How about **`a = tan(a) + z`**

7) As a continuation of exercise 6, let's try some additional Julia Set functions. Set **axrng = 2.0**, **hcenter = 0**, and **vcenter = 0**. Make the following modifications to the appropriate portion of the **def drawjulia():** function.

```
        while n < 200:
             n+=1
             a = exp(a) + z
             zz = abs(a)
             if zz > 200:
                  glColor3f(cos(zz),sin(zz)*cos(zz),sin(zz))
                  glVertex2f(x,y)
                  n = 5001
        if zz < 200:
             glColor3f(tan(n),sin(zz),cos(n))
             glVertex2f(x,y)
```

Notice the change in the conditional values in the **while** loop. We may be able to use smaller escape time values. You can experiment with escape time if you wish. The second **glColor3f** statement has also been changed. The new Julia Set equation is **a = exp(a) + z**. This is the exponential function. The resulting plot from these modifications is shown in figure **Exercise 7**.

8) Try this modification, keeping **axrng**, **hcenter**, and **vcenter** the same as in exercise 7:

```
   while n < 200:
        n+=1
        a = a**3 + z
        zz = abs(a)
        if zz > 2:
             glColor3f(cos(zz),sin(zz)*cos(zz),sin(zz))
             glVertex2f(x,y)
             n = 5001
   if zz < 200:
        glColor3f(tan(zz),sin(zz),cos(n))
        glVertex2f(x,y)
```

The resulting plot is found in figure **Exercise 8** below. By now you should have the idea that the number of different possibilities is infinite! Explore! Create! Don't be afraid to try something new and/or different.

9) A rather beautiful Julia Set can be found by using the complex number (-0.780737, -0.105882i). You can see this set by changing the Julia Set seed to:

```
   z = complex(-0.780737,-0.105882)
```

and the equation back to **a = a**2 + z**. Also, comment out the **glColor3f** and the **glVertex2f** statements under the **if z > 2:** statement and modifying the **glColor3f** statement under the **if z < 2:** statement as follows:

```
   glColor3f(3/sin(3*zz),cos(3*z.real),2*sin(zz))
```

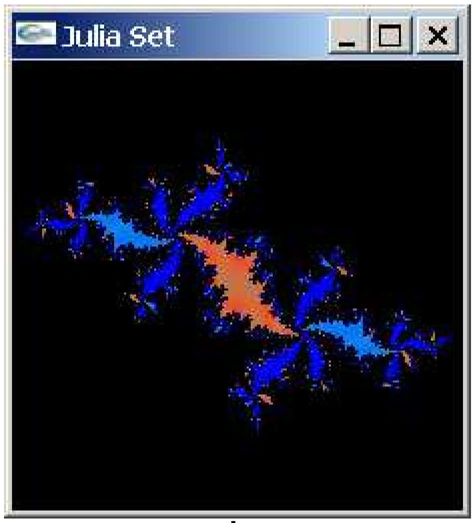
The resulting plot can be seen in figure **Exercise 9**. You might need to change the number of iterations to **if n < 200:** or higher to get the best plot.

10) There is another method for creating a Julia Set. This method is called the inverse iteration method. Until this exercise, we have used a Julia Set complex number seed and iterated each pixel point in the graphics window to see if the iteration escapes to infinity or not. The general equation we used was $a = a^2 + z$ where **z** was the constant complex seed used to generate the Julia Set. We can also generate a Julia Set by taking the inverse of this equation; the complex square root. In this iteration method, the pixel points rapidly converge to the Julia Set. The advantages of the inverse iteration method are that we can quickly generate a Julia Set. The disadvantage is that the Julia Set is not filled and can be somewhat incomplete. The listing for an inverse iteration method is as follows:

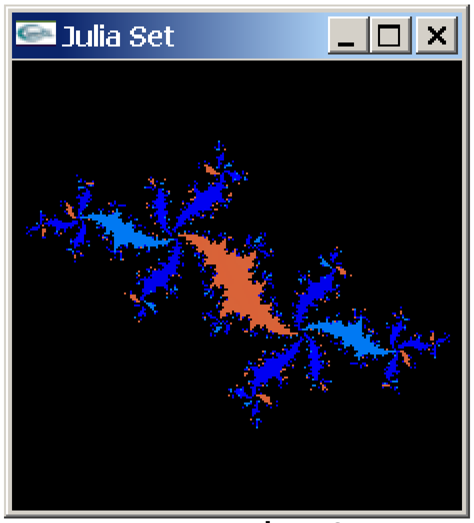
```
# PyInverseJulia.py
# Plot a Julia set
# Using inverse iteration

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from random import *
from Numeric import *
import sys

# If psyco isn't installed, delete the next two lines!
import psyco
psyco.full()

axrng = 2.0
width = 400
height = 400

def init():
   glClearColor(0.0, 0.0, 0.0, 0.0)
   gluOrtho2D(-axrng,axrng,-axrng,axrng)

def drawinvjulia():
   glClear(GL_COLOR_BUFFER_BIT)

   # complex seed point... same as exercise 9
   a = complex(-0.780737,-0.105882)

   # plot 10000 points, one for each iteration
   for i in range(0,10000):
         glBegin(GL_POINTS)
         x = 2*axrng*random()-2
         y = 2*axrng*random()-2

         n = 0
```

```
        z = complex(x,y)

        # since there are two square roots
        # we randomly choose between them
        while n < 10:
             n+=1
             if random() < 0.5:
                  z = sqrt(z-a)
             else:
                  z = -sqrt(z-a)
             zz = abs(z)

        glColor3f(3/sin(3*zz),cos(3*z.real),2*sin(zz))
        glVertex2f(z.real,z.imag)
        glEnd()
        glFlush()

def main():
   glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE)
   glutInitWindowPosition(50, 50)
   glutInitWindowSize(width, height)
   glutInit(sys.argv)
   glutCreateWindow("Julia Set")
   glutDisplayFunc(drawinvjulia)
   init()
   glutMainLoop()

main()

# End Program
```

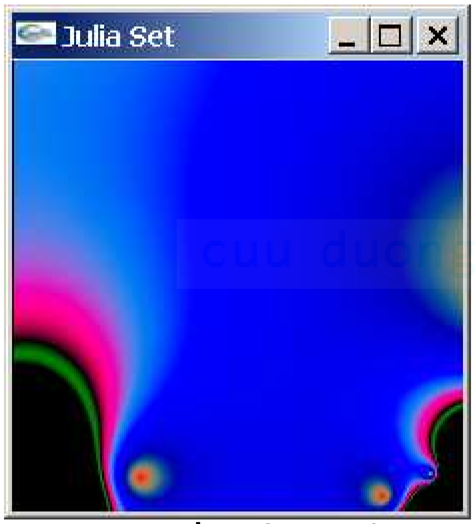
Figure **`Exercise 10`** illustrates this program. You may notice some stray points inside the Julia Set. You can eliminate these points at the expense of program speed by increasing the value in the **`while n < 10:`** loop to **`while n < 50:`**. Try using different complex seeds and changing the number iterations (and points!). You can also experiment with coloration and see the effects on the plot.



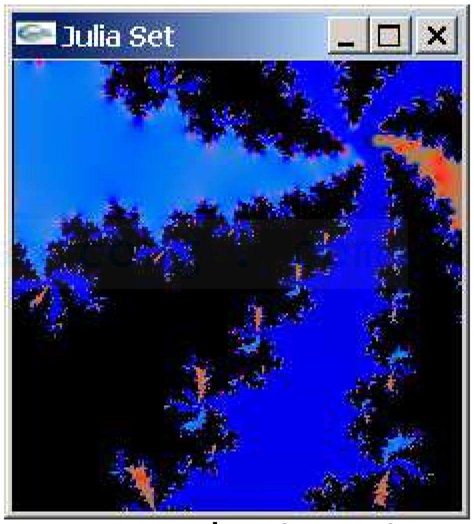
Exercise 1



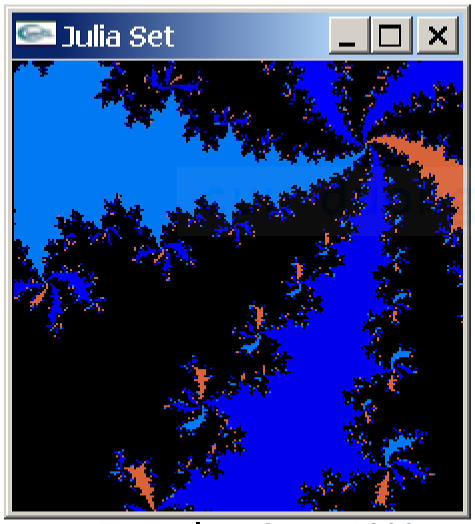
Exercise 2



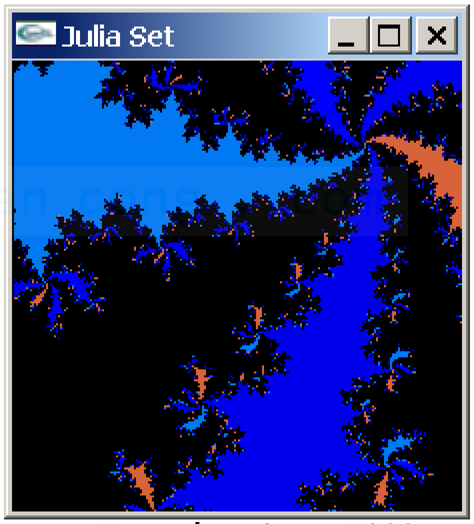
Exercise 3: n<10



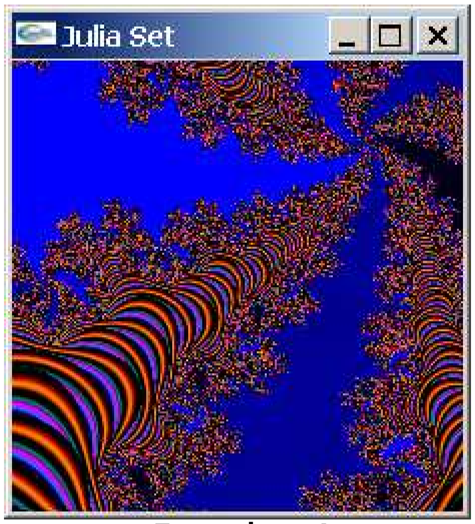
Exercise 3: n<50



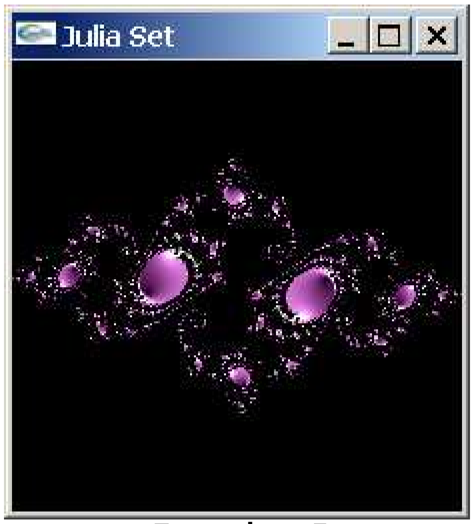
Exercise 3: n<1000



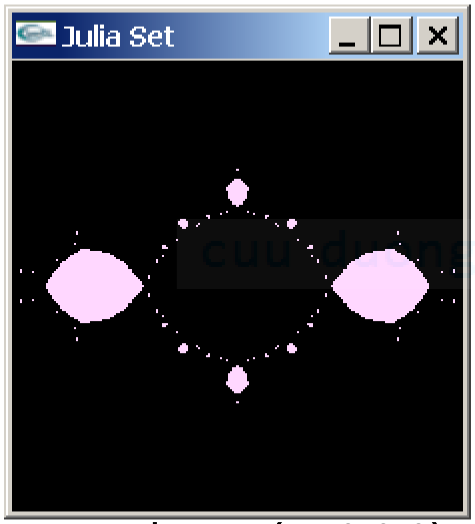
Exercise 3: n<5000



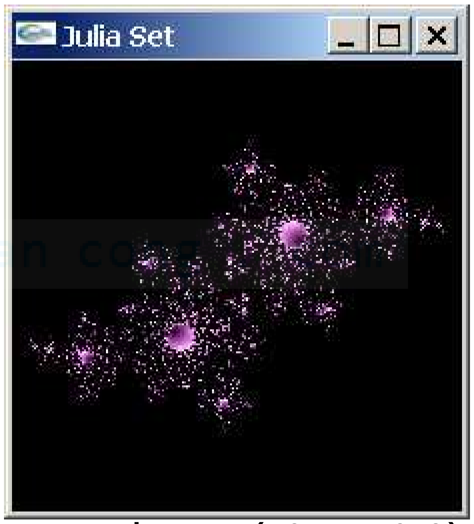
Exercise 4



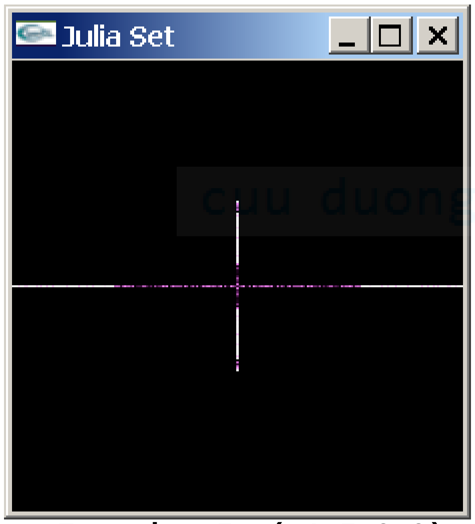
Exercise 5a



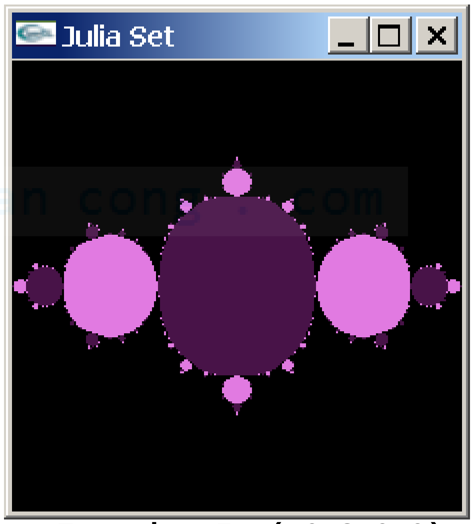
Exercise 5: (-1.0,0.0)



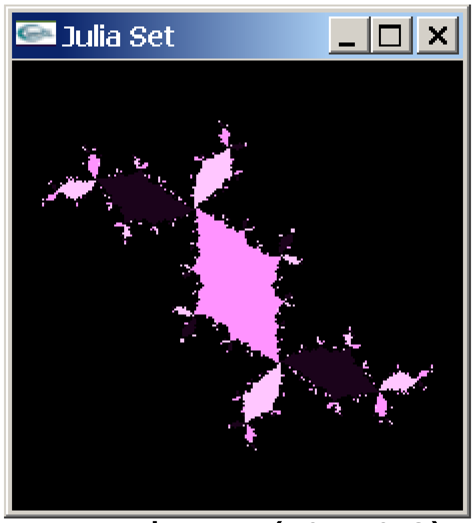
Exercise 5: (-0.4, -0.6)



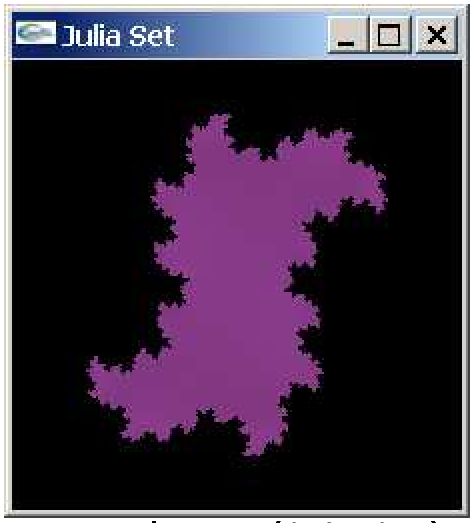
Exercise 5: (-1.5,0.0)



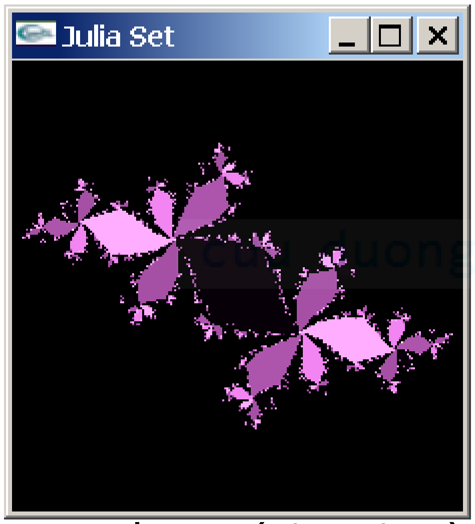
Exercise 5: (-0.8,0.0)



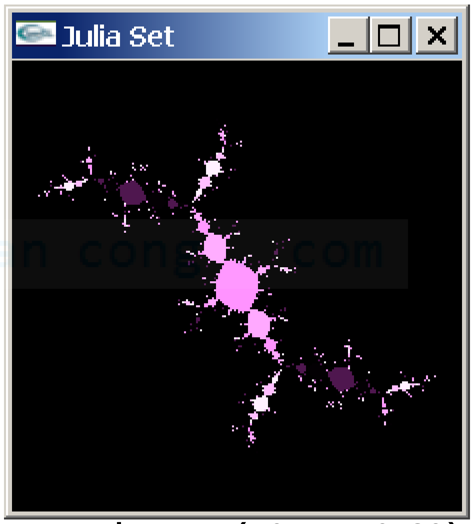
Exercise 5: (-0.1,0.8)



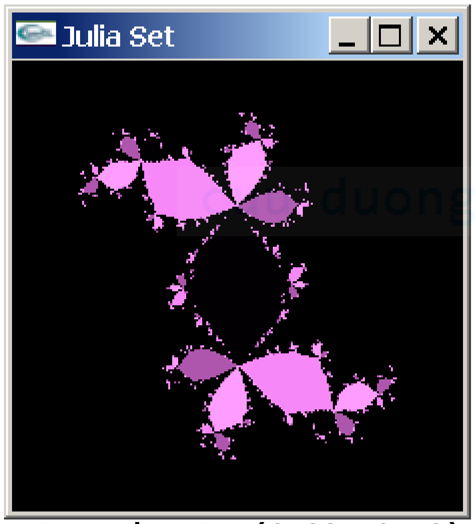
Exercise 5: (0.3,-0.4)



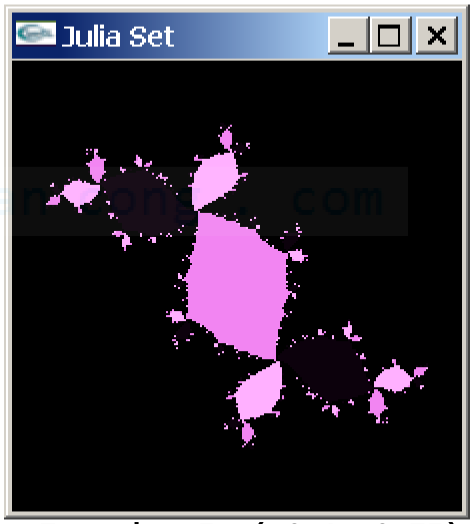
Exercise 5: (-0.5, 0.57)



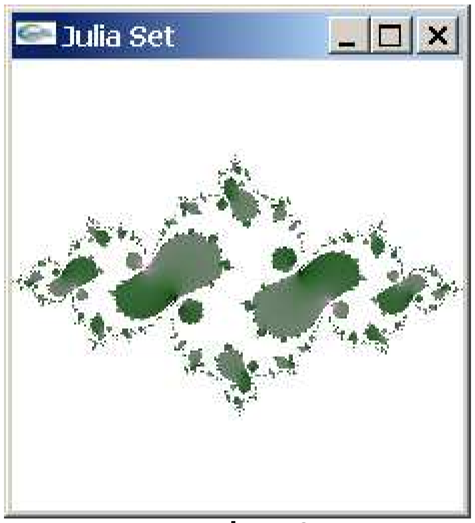
Exercise 5: (-0.11, 0.86)



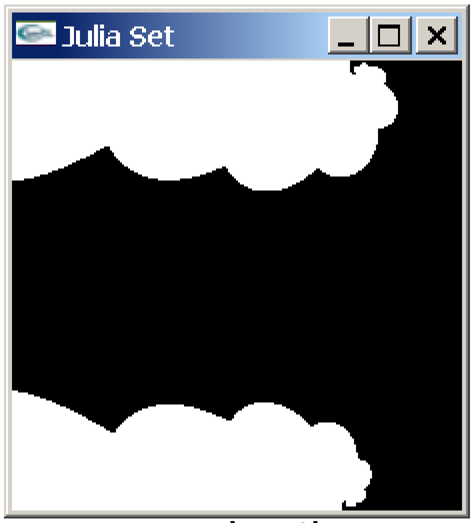
Exercise 5: (0.28, 0.53)



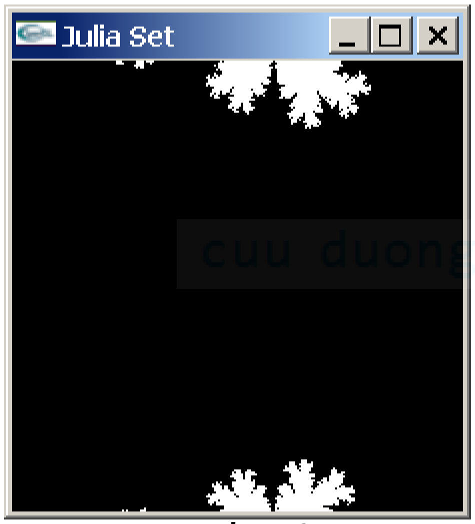
Exercise 5: (-0.1, 0.75)



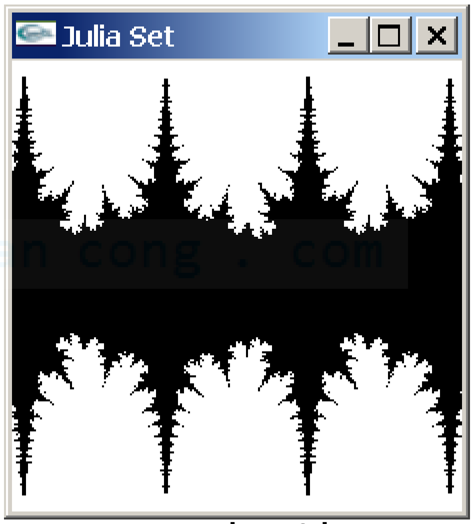
Exercise 6a



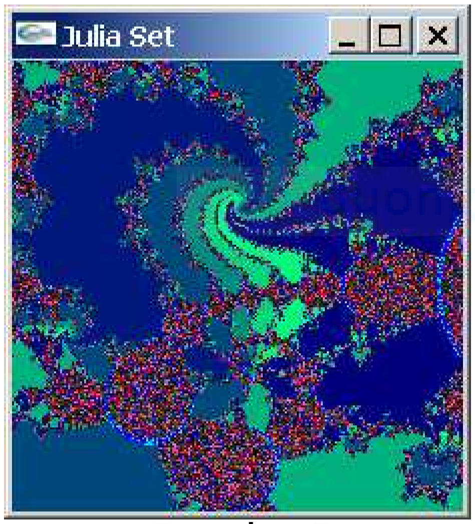
Exercise 6b



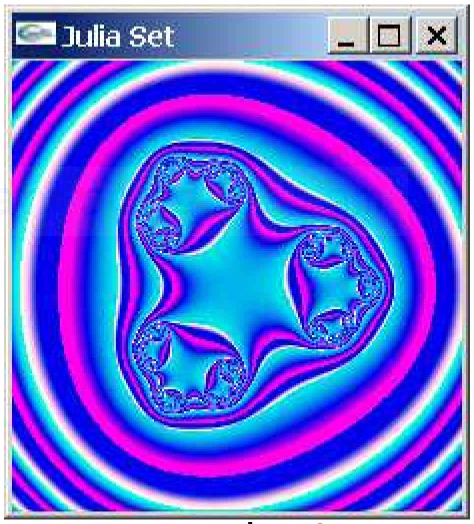
Exercise 6c



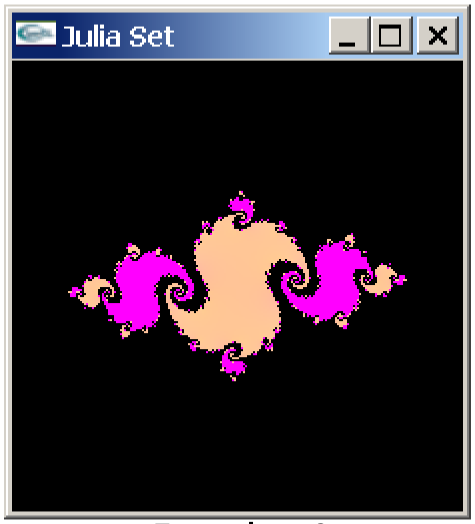
Exercise 6d



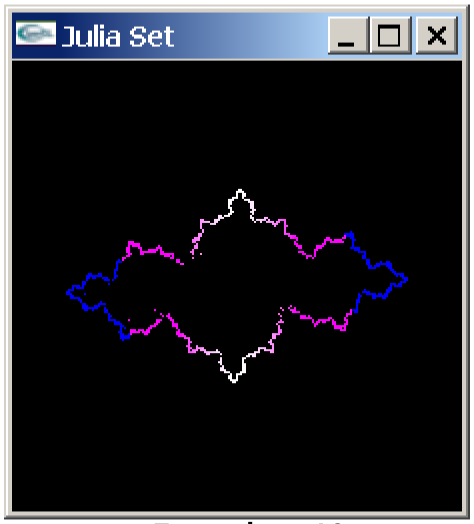
Exercise 7



Exercise 8



Exercise 9



Exercise 10

Section 7.7 Explorations with the Mandelbrot Set

In the last section we explored the basics of the Julia Set and the iterated function, **`a = a**2 + z,`** which leads to the determination of how the set will be plotted. In this section, we are going to explore the Mandelbrot Set (M-Set), the parent of the Julia Set family, on two levels. First, we are going to see how the M-Set is created, and second, we are going to include a "cheesy" mouse routine[48] that will hopefully allow us to explore the M-Set "live" without having to change the **`hcenter`**, **`vcenter`**, and **`axrng`** values prior to each program execution.

The M-Set is similar to the Julia Set in that we are going to iterate all the complex numbers found in the graphics window pane to see if those numbers escape to infinity or remain constrained. The difference is that with the Julia Set, we chose an initial complex number seed and then iterated each pixel after converting the coordinates of that pixel to a complex number. With the M-Set, we are not going to pre-choose an initial complex number seed. Rather, we are going to use each pixel in the graphics window as its own seed and plot the results. Based on this reasoning, since each pixel in the graphics window is a unique complex number, the M-Set is a catalog of all possible Julia Sets, one unique Julia Set for each pixel![49] This makes the M-Set infinitely complex and certainly worthy of our attention. The M-Set is also quite beautiful regardless of the level of magnification. Before we go any further, though, take the time to Google the Mandelbrot Set on the web. You'll find a multitude of sources, so look at only a few. I would recommend choosing sites that have ".edu" endings. As you'll discover (or have already discovered), the M-Set is named after Benoit Mandelbrot, who wrote one of the pioneering texts on fractals.[50]

Here is the listing of the M-Set program for this section. It is a bit more involved than previous listings because it includes some additional mouse and keyboard functions. After the listing I'll attempt to clarify the new sections of code.

```
# PyMandelBrot.py
# Plot a Mandelbrot set
# And include a mouse zoom

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

# If psyco isn't installed, delete the next two lines!
import psyco
psyco.full()

# Set initial window width and height
# Declare global variables
```

[48] I can hear the groans. Yes, pun intended.
[49] We will attempt to demonstrate this idea in an exercise.
[50] Mandelbrot, B. (1982). "The Fractal Geometry of Nature". W. H. Freeman

```
global width
global height
global hcenter
global vcenter
global axrng
global hstep
global vstep
global yinit
global xinit

width = 400
height = 400

def zap():
     global hcenter
     global vcenter
     global axrng
     hcenter = 0.0
     vcenter = 0.0
     axrng = 2.0
     init()

def init():
     # Identify the globals
     global hcenter
     global vcenter
     global axrng
     global hstep
     global vstep
     global yinit
     global xinit
     global yfinal
     global xfinal

     # Set the screen plotting coordinates and the step
     glClearColor(0.0, 0.0, 0.0, 0.0)
     hstep = 2*axrng/(width)
     vstep = 2*axrng/(height)
     yinit = vcenter + axrng
     xinit = hcenter - axrng
     yfinal = vcenter - axrng
     xfinal = hcenter + axrng

     #  Fill the entire graphics window!
     glViewport(0, 0, width, height)

     #  Set the projection matrix... our "view"
     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     # Set the window plot coordinates
     gluOrtho2D(xinit,xfinal,yfinal,yinit)
```

```
    #  Set the matrix for the object we are drawing
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    glutPostRedisplay()

def keyboard(key, x, y):
    #  Allows us to quit by pressing 'Esc' or 'q'
    if key == chr(27):
        sys.exit()
    if key == "z":
        zap()
    if key == "q":
        sys.exit()

def drawmandel():
  glClear(GL_COLOR_BUFFER_BIT)

  y = yinit
  while y > yfinal:
    y -= vstep
    x = xinit
    while x < xfinal:
      x += hstep

      n = 0
      z = a = complex(x,y)
      glBegin(GL_POINTS)

      # n < 200 is the number of iterations
      # Increase this value to show finer detail
      # However finer detail results in slower execution
      while n < 200:
        n+=1
        z = z**2 + a
        zz = abs(z)

        # zz > 2 is the critical escape value
        # Some functions require larger escape values
        # This zz > 2 conditional provides coloration for
        # points outside the M-Set set
        if zz > 2:
          # Weird colors around the M-Set
          #glColor3f(3*sin(3*z.real),cos(3/z.real),4*cos(zz))
          #glVertex2f(x,y)
          n = 5001

      # This zz < 2 conditional provides coloration for
      # points inside the M-Set.
      if zz < 2:
        # Coloration in the M-Set
        #glColor3f(3/sin(3*zz),cos(3*z.real),2*sin(zz))
```

```
            glColor3f(0.9, 0.2, 0.5)
            glVertex2f(x,y)
         glEnd()
      glFlush()

def mouse(button, state, x, y):
      global hcenter
      global vcenter
      global axrng

      # Detect the left/right mouse buttons and the click
      # Followed by resetting the origin
      # Left mouse button zooms in, right button zooms out
      if button == GLUT_LEFT_BUTTON and state == GLUT_DOWN:
            axrng = axrng/2
      if button == GLUT_RIGHT_BUTTON and state == GLUT_DOWN:
            axrng = 2*axrng
      if state == GLUT_DOWN:
            hcenter = xinit + (xfinal - xinit)*x/width
            vcenter = yinit + (yfinal - yinit)*y/height
            print hcenter, vcenter
            init()

def main():
      glutInit(sys.argv)
      glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE)
      glutInitWindowPosition(50, 50)
      glutInitWindowSize(width, height)
      glutCreateWindow("Mandelbrot Set")
      glutDisplayFunc(drawmandel)
      glutMouseFunc(mouse)
      glutKeyboardFunc(keyboard)
      zap()
      glutMainLoop()

main()

# End Program
```

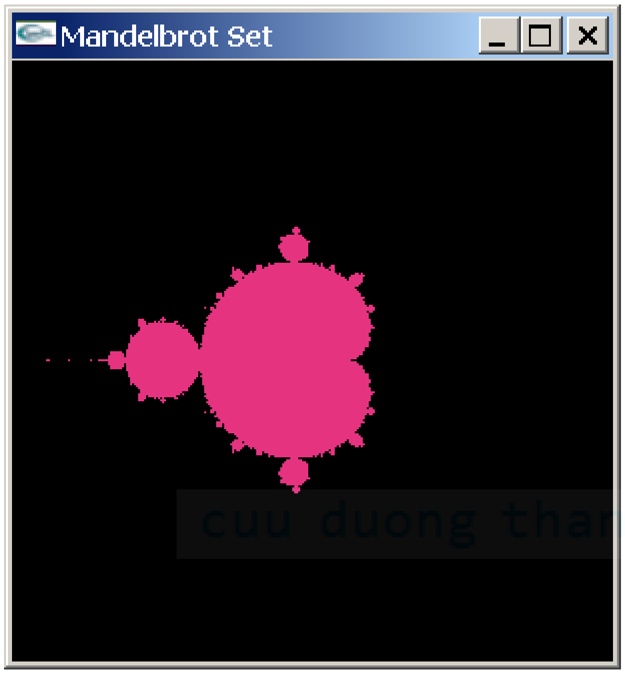
Before any explanations are given, let's look at the one-color plot shown in figure **M-Set** on the next page. If you haven't already done so, run the program. Remember to be patient. Fractals take time to draw! Notice the shape of the M-Set. Do you see a Cardiod? Now left-click your mouse pointer on a point on the edge of the M-Set and see if you can display a zoomed version. Keep zooming until you see vertical and/or horizontal line artifacts.[51] Now right-click on a portion of the graph to zoom out again. See how small you can make the M-Set while still retaining an identifiable shape. If you want to return to the original state of the M-Set, make certain the graphics window has focus[52] and press the "z" key (for "zap"). Explore the M-Set by clicking on interesting

[51] We'll try to fix these artifacts in an exercise.
[52] Click on the caption bar or make certain the caption bar is highlighted.

portions of the graph. Can you find M-Set miniatures hiding within the edges of the full-size Mandelbrot Set? This plot was created by using **`glColor3f(0.9, 0.2, 0.5)`**. If you comment this statement and uncomment:

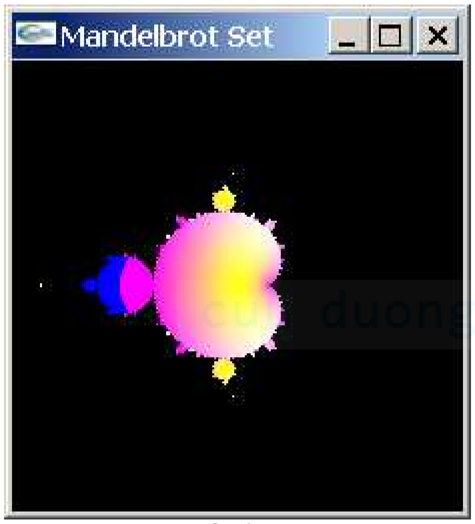
```
glColor3f(3/sin(3*zz),cos(3*z.real),2*sin(zz))
```

you'll see something like figure **`M-Set a`** below. Uncommenting the lines under **`# Weird colors around the M-Set`** statement will result in figure **`M-Set b`**. Nice!

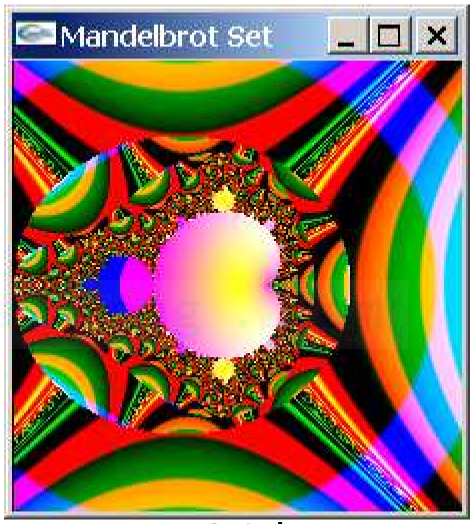


M-Set

One color is OK… but look below!



M-Set a



M-Set b

We start the M-Set program with the usual gang of import statements, followed by a few global variables as shown below:

```
# Set initial window width and height
# Declare global variables
global width
global height
global hcenter
global vcenter
global axrng
global hstep
global vstep
global yinit
global xinit

width = 400
height = 400
```

We are going to actually declare these variables as **`global`** because we want to have the capability of changing their values during the program execution. The initial window **`width`** and **`height`** is set to 400 as in the Newton's Method and Julia Set programs, but you may need to make this a bit smaller (or larger) depending on your display resolution.

The first function encountered is "called" from within the **`def main():`** function and is named **`def zap():`**

```
def zap():
      global hcenter
      global vcenter
      global axrng
      hcenter = 0.0
      vcenter = 0.0
      axrng = 2.0
      init()

      # End Function
```

The purpose of **`def zap():`** is to provide a method of returning a zoomed M-Set back to its original state by setting both **`hcenter`** and **`vcenter`** back to 0.0 and setting **`axrng = 2.0`** or its original value. The last line of **`def zap():`** calls the **`def init():`** function just as if we've restarted the program.

```
def init():
      # Identify the globals
      global hcenter
      global vcenter
      global axrng
      global hstep
      global vstep
      global yinit
      global xinit
      global yfinal
```

```
        global xfinal
        global count

        # Set the screen plotting coordinates and the step
        glClearColor(0.0, 0.0, 0.0, 0.0)
        hstep = 2*axrng/(width+1)
        vstep = 2*axrng/(height+1)
        yinit = vcenter + axrng
        xinit = hcenter - axrng
        yfinal = vcenter - axrng
        xfinal = hcenter + axrng

        #  Fill the entire graphics window!
        glViewport(0, 0, width, height)

        #  Set the projection matrix... our "view"
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()

        # Set the window plot coordinates
        gluOrtho2D(xinit,xfinal,yfinal,yinit)

        #  Set the matrix for the object we are drawing
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
        glutPostRedisplay()

        # End Function
```

The **def init():** function first declares the global variables[53] found at the beginning of the program and defines two additional globals, **yfinal** and **xfinal**. After declaring the global variables can we clear the screen. The next six lines calculate the **hstep** and **vstep** for the current window based on the **axrng** (multiplied by 2 to encompass the entire range from **–axrng** to **+axrng**) divided by the appropriate (**width + 1)** and (**height + 1)** values. Adding 1 to **width** and **height** helps alleviate some of the glitches or vertical lines that appear in the graphics window when zooming. We then calculate the initial and final **x** and **y** values (**xinit, yinit, xfinal, yfinal**) for the **gluOrtho2D** statement based on the current **hcenter**, **vcenter**, and **axrng** settings. Think about this! Read the preceding sentences again until you understand what is happening with the graphics display! The remainder of the function sets the viewport and the virtual window dimensions. The final **glutPostRedisplay()** causes a screen refresh using the new settings. The value of this **def init():** function is in its ability to redefine the screen viewing dimensions when zooming, while repositioning the center of the viewing window using the mouse.[54] It is the screen redefinition calculations and their use in **gluOrtho2D** that allow us to zoom into the M-Set.

[53] This is necessary IF we want to change the values of the variables in the **def init():** function.
[54] See the **def mouse(button, state, x, y):** function section.

The **keyboard** function has been expanded slightly to allow us the ability to "zap" all the global variables back to their initial values, thereby returning the M-Set to its original state.

```
def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'
     if key == chr(27):
          sys.exit()
     if key == "z":
          zap()
     if key == "q":
          sys.exit()

     # End Function
```

The "**z**" key calls the **def zap():** function, which resets the global variables and then calls the **def init():** function as described above. The **def init():** function then sets the viewing window to its initial coordinates and refreshes the screen with the original M-Set, again as described previously.

The **def drawmandel():** function is very similar to the def **drawjulia():** function in the previous section of the text. The differences are that we do NOT define a particular complex number prior to entering the **while y > yfinal:** loop. Remember that in calculating the M-Set we are testing EVERY pixel point in the graphics window to see if that point escapes to infinity or not. Therefore we do not require or want an initial complex number seed to use in our iteration loop. The critical statement then becomes:

```
     z = a = complex(x,y)
```

This double assignment statement sets both **z** and **a** to the same complex number initially, based on the current **x** and **y** coordinates. The program then enters the **while n** iteration loop where **a** is kept constant (**a** is the current point we are testing) and **z** is iterated in the **z = z**2 + a** statement to see if it tends to escape toward infinity or not. If after an arbitrary number of iterations **(n < 200** in this example program) the distance (modulus) of **z** (calculated by **zz = abs(z)**) is greater than 2, we know the point is outside the M-Set and we can color it accordingly (or not). If after 200 iterations **zz < 2**, we know the point is within the M-Set and we can color those points (or not) accordingly. The M-Set is actually the boundary between these two regions, the exterior region where all points escape to infinity and the interior region where all points are constrained. The M-Set is on the edge of chaos, so to speak!

The function which handles the mouse behavior is new and is properly identified in **def main():** using the **glutMouseFunc(mouse)** statement. It is the **mouse** function that allows us to choose a pixel for zooming and repositioning the M-Set. The "chosen pixel" has **x** and **y** coordinates which are sent to the **mouse** function for processing. The mouse function is shown and explained below:

```
def mouse(button, state, x, y):
     global hcenter
```

```
        global vcenter
        global axrng

        # Detect the left/right mouse buttons and the click
        # Followed by resetting the origin
        # Left mouse button zooms in, right button zooms out
        if button == GLUT_LEFT_BUTTON and state == GLUT_DOWN:
             axrng = axrng/2
        if button == GLUT_RIGHT_BUTTON and state == GLUT_DOWN:
             axrng = 2*axrng
        if state == GLUT_DOWN:
             hcenter = xinit + (xfinal - xinit)*x/width
             vcenter = yinit + (yfinal - yinit)*y/height
             print hcenter, vcenter
             init()

# End Function
```

The **def mouse(button, state, x, y):** statement contains the name[55] of the function and some variables or parameters within the parentheses. These variables must always be present in the **mouse** function defined with the **glutMouseFunc()** statement, although you may choose to use different variable names. The first variable, **button**, stores the specific mouse button used to trigger or call the mouse function. Possible values are **GLUT_LEFT_BUTTON**, **GLUT_RIGHT_BUTTON**, or **GLUT_MIDDLE_BUTTON** depending on which mouse button was clicked to call the mouse function. The second variable, **state**, holds the status of the mouse button. The possible values are **GLUT_DOWN** and **GLUT_UP**. Each state will trigger or call the mouse function.[56] This means that you can perform a specific operation when the mouse button is clicked (**GLUT_DOWN**) and another specific operation when the mouse button is released (**GLUT_UP**). The final two variables, **x** and **y**, store the window pixel coordinates of the mouse pointer when the mouse button is clicked.

Within the **def mouse(button, state, x, y):** function are three conditional statement blocks. The first:

```
        if button == GLUT_LEFT_BUTTON and state == GLUT_DOWN:
             axrng = axrng/2
```

is executed when the LEFT mouse button state is the DOWN position.[57] The second conditional statement:

```
        if button == GLUT_RIGHT_BUTTON and state == GLUT_DOWN:
             axrng = 2*axrng
```

[55] Again, we don't have to call the **mouse** function "**mouse**". We can name it whatever we please as long as we use the name in the **glutMouseFunc()** statement.
[56] If you forget this fact, some interesting behavior can result. If you do not want an event to occur when the mouse button is released, place the desired "mouse-triggered" behavior within the **GLUT_DOWN** conditional block of code as shown in this program.
[57] How do you know this?

catches the right mouse button click.
The third conditional statement:

```
if state == GLUT_DOWN:
     hcenter = xinit + (xfinal - xinit)*x/width
     vcenter = yinit + (yfinal - yinit)*y/height
     print hcenter, vcenter
     init()
```

is executed ONLY when one of the mouse buttons is clicked.

In either case, left or right button, new screen center coordinates are calculated based on the old screen coordinate ranges and the current mouse pointer position coordinates. The intent of the left mouse button click is to zoom in on the M-Set, so we halve the **axrng** variable. If this seems contrary to common sense, think of it as a microscope. When we move to higher magnifications, we are looking at a much smaller field of view. If the magnification power is doubled, the field of view is halved. We are accomplishing the same effect here by halving the **axrng** field of view, thereby doubling the magnification.

The right button click allows us to zoom out by doubling the size of the **axrng** variable (doubling the field of view halves the magnification). If either mouse button is clicked (**if state == GLUT_DOWN** is true), we calculate the new **hcenter** and **vcenter** origin and then **def init():** is called to implement the changes. The **print hcenter, vcenter** statement prints the current screen center coordinates in the console window for reference (these coordinates would correspond to Julia Set "seeds"). This **print** statement line is not crucial to the operation of the program, but may be useful for debugging purposes.

The **def main():** function is similar to previous programs with the exception of the **glutMouseFunc(mouse)** statement.

When you look at the M-Set you are literally looking at infinity. It is possible (theoretically) to magnify the M-Set until it is the size of the known universe… and even then the detailed swirls never end. You can still zoom further! Literally mountains of papers and texts have been written about this beautiful mathematical object and still we do not know everything there is to know about the M-Set fractal. Countless hours of computer time have been spent[58] plotting the M-Set on everything from super-computers to hand held devices. When you explore the M-Set by zooming into specific areas it is quite possible that after several magnifications you will be viewing something never before seen by human eyes. I think you will agree that this is a remarkable object indeed!

As stated previously, the M-Set represents a catalog of Julia Sets. If you recall, we had to explicitly provide a complex number seed to the Julia Set program in order to calculate and plot a Julia Set. The M-Set, in contrast, is created by testing every point in the complex plane (well… at least every point represented by a pixel and within the **axrng** boundaries) and plotting those points based on whether or not the iterations

[58] Some might say “wasted”, but it depends on your point of view and who is paying your salary.

escape to infinity or stay bounded. Each pixel in the M-Set thus represents a complex seed for a unique Julia Set. It would be interesting if we could simply point and click on a screen pixel in and around the M-Set and plot the Julia Set for that pixel. It would also be nice to have the capability of zooming into and out of the Julia Set and then return to the same M-Set from which we started. Such a program would be more complex than anything we've done thus far, but that shouldn't be a hindrance. Using the M-Set program from this section as a skeleton, compare it to the following listing and make the changes where necessary. Make certain you save this program under a new name so that you can preserve the old M-Set program! Comments have been liberally applied to the code to explain the purpose of each section of program.

```
# PyMandelJulia.py
# Plot a Mandelbrot set
# And include a mouse zoom with
# Julia Set option enabled

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

# If psyco isn't installed, delete the next two lines!

import psyco
psyco.full()

# Set initial window width and height
# Declare global variables

global width
global height
global hcenter
global vcenter
global axrng
global hstep
global vstep
global yinit
global xinit

# The following globals allow for the
# Julia Set options

global mandel
global tmprng
global julhcenter
global julvcenter
global julflag
global julx
global july
```

```
width = 400
height = 400

# mandel = 1 for M-Set
# mandel = 0 for Julia Set
# Start with the M-Set

mandel = 1

def zap():
     # Reset everything

     global hcenter
     global vcenter
     global axrng
     global mandel
     global julflag
     hcenter = 0.0
     vcenter = 0.0
     axrng = 2.0
     mandel = 1
     julflag = 0
     init()

def init():
     # Identify the globals

     global hcenter
     global vcenter
     global axrng
     global hstep
     global vstep
     global yinit
     global xinit
     global yfinal
     global xfinal
     global mandel
     global julhcenter
     global julvcenter

     # Set the screen plotting coordinates and the step

     glClearColor(0.0, 0.0, 0.0, 0.0)

     # Dividing by (width+1) and (height+1) delays
     # the onset of screen glitches or artifacts when
     # zooming by making the hstep and vstep slightly smaller

     hstep = 2*axrng/(width+1)
     vstep = 2*axrng/(height+1)

     # if mandel == 1 then we are plotting the M-Set
```

```
	if mandel == 1:
		yinit = vcenter + axrng
		xinit = hcenter - axrng
		yfinal = vcenter - axrng
		xfinal = hcenter + axrng
	else:
		# if mandel <> 1 then we plot Julia
		# the Julia Set uses different
		# global variables so we don’t forget
		# the M-Set parameters

		yinit = julvcenter + axrng
		xinit = julhcenter - axrng
		yfinal = julvcenter - axrng
		xfinal = julhcenter + axrng

	#  Fill the entire graphics window!

	glViewport(0, 0, width, height)

	#  Set the projection matrix... our "view"

	glMatrixMode(GL_PROJECTION)
	glLoadIdentity()

	# Set the window plot coordinates

	gluOrtho2D(xinit,xfinal,yfinal,yinit)

	#  Set the matrix for the object we are drawing

	glMatrixMode(GL_MODELVIEW)
	glLoadIdentity()
	glutPostRedisplay()

def keyboard(key, x, y):
	global mandel
	global tmprng
	global axrng
	global hcenter
	global vcenter
	global julflag

	if mandel == 1:
		# We are working with the M-Set
		# Store the M-Set axrng zoom factor
		# So that we can restore it later if needed

		tmprng = axrng

	#  Allows us to quit by pressing 'Esc' or 'q'
```

```
        if key == chr(27):
             sys.exit()

        if key == "z":
             zap()

        if key == "j":
             # Toggle the Julia Set
             # and set axrng to original zoom

             mandel = 0
             axrng = 2.0

        if key == "m":
             # Toggle M-Set and restore the last
             # M-Set axrng value so the M-Set looks
             # the same as it did when we left it

             mandel = 1
             axrng = tmprng
             julflag = 0
             init()

        if key == "q":
             sys.exit()

def drawmandel():
        glClear(GL_COLOR_BUFFER_BIT)
        y = yinit

        # toggle Julia Set or M-Set
        # mandel == 0 is the Julia Set
        # julx and july contain the Julia Set
        # seed coordinates

        if mandel == 0:
             a = complex(julx, july)

        while y > yfinal:
          y-= vstep
          x = xinit
          while x < xfinal:
             x+= hstep

             n = 0

             # Choose M-Set or Julia Set
             # If Julia Set is toggled, “a” already contains
             # the complex number seed

             if mandel == 1:
                  z = a = complex(x,y)
```

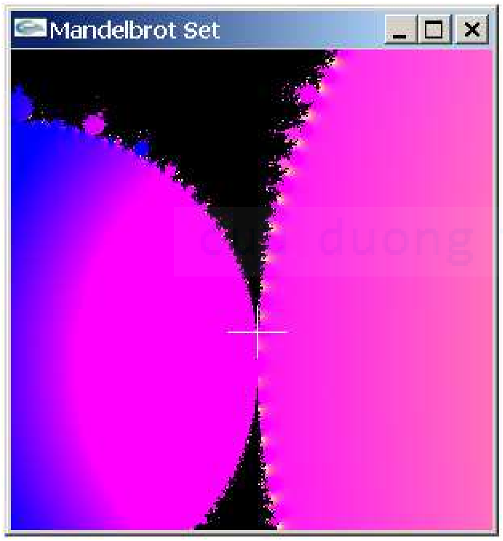
```
            else:
                 z = complex(x,y)

            glBegin(GL_POINTS)

            # Escape time… increase this value above 25
            # for a more detailed plot.  Decrease
            # this value for more speed.

            while n < 25:
               n+=1
               z = z**2 + a
               zz = abs(z)

               # This is the escape distance.  For some
               # M-Set/Julia Sets such as sin() or exp() you
               # may need to set this value higher than 2
               # 50 works well for sin() functions

               if zz > 2:
                  # Weird colors outside the M-Set
                  #glColor3f(3*sin(3*z.real),cos(3*z.real),4*cos(zz))
                  #glVertex2f(x,y)

                  n = 5001

            # The same goes for this zz < 2 statement as above

            if zz < 2:
               # Coloration inside the M-Set

               glColor3f(3*sin(3*zz),cos(3*z.real),2*sin(zz))
               glVertex2f(x,y)

            glEnd()
         glFlush()

def mouse(button, state, x, y):
     global hcenter
     global vcenter
     global axrng
     global julhcenter
     global julvcenter
     global julflag
     global julx
     global july

     # Detect the left/right mouse buttons and the click
     # Followed by resetting the origin
     # Left mouse button zooms in, right button zooms out

     if button == GLUT_LEFT_BUTTON and state == GLUT_DOWN:
```

```
if mandel == 1:
   hcenter = xinit + (xfinal - xinit)*x/width
   vcenter = yinit + (yfinal - yinit)*y/height
   axrng = axrng/2
   init()
else:
   # We use different center point variables here
   # to keep the Julia Set and M-Set calculations
   # separate

   julhcenter = xinit + (xfinal - xinit)*x/width
   julvcenter = yinit + (yfinal - yinit)*y/height

   # We use a flag variable here so that the first
   # Julia Set plot is normal size regardless of the
   # Zoom factor on the M-Set.  We don’t want to
   # cause a zoom on the first Julia Set

   if julflag == 0:
      # Print the value of the Julia Set seed

      print "Julia", julhcenter, julvcenter

      # Store the pixel coordinates in julx and july
      # for the Julia Set seed

      julx = julhcenter
      july = julvcenter

      # Set the following variables to zero
      # so the first Julia Set is centered in
      # the graphics display window

      julhcenter = 0.0
      julvcenter = 0.0

      # Show the Julia Set!
      init()

   else:
      # NOW we can zoom on the Julia Set

      julhcenter = xinit + (xfinal - xinit)*x/width
      julvcenter = yinit + (yfinal - yinit)*y/height
      axrng = axrng/2
      init()

   # Set the flag so subsequent mouse clicks zoom
   # into the Julia Set

   julflag = 1
```

```
    if button == GLUT_RIGHT_BUTTON and state == GLUT_DOWN:
        # This section is similar to the previous
        # Section except that here we zoom out!

        if mandel == 1:
           hcenter = xinit + (xfinal - xinit)*x/width
           vcenter = yinit + (yfinal - yinit)*y/height
           axrng = 2*axrng
           init()

        else:
           julhcenter = xinit + (xfinal - xinit)*x/width
           julvcenter = yinit + (yfinal - yinit)*y/height

           # Again, we don't want to initially zoom into
           # the Julia Set... we want a "normal" Julia First
           if julflag == 0:
              print "Julia", julhcenter, julvcenter
              julx = julhcenter
              july = julvcenter
              julhcenter = 0.0
              julvcenter = 0.0
              init()

           else:
              julhcenter = xinit + (xfinal - xinit)*x/width
              julvcenter = yinit + (yfinal - yinit)*y/height
              axrng = 2*axrng
              init()

           julflag = 1

def main():
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE)
    glutInitWindowPosition(50, 50)
    glutInitWindowSize(width, height)
    glutCreateWindow("Mandelbrot Set")
    glutDisplayFunc(drawmandel)
    glutMouseFunc(mouse)
    glutKeyboardFunc(keyboard)
    zap()
    glutMainLoop()

main()

# End Program
```

This program functions in much the same fashion as the previous M-Set program with the exception of the added Julia Set option. To display the Julia Set of any pixel point in the M-Set graphics window, press the "**`j`**" key on the keyboard and then click on the pixel using either mouse button. Once the Julia Set has displayed, you can then zoom in or out just as with the M-Set program. To return the M-Set you were viewing prior to the Julia Set, press the "**`m`**" key. Remember that EACH pixel[59] in the M-Set graphics window represents a different Julia Set. You could spend a LOT of time with this program exploring the various Julia Sets associated with different locations in the M-Set!

An example of this program's execution is shown in figures **`M-Set`** and **`J-Set`** below and on the next page. An escape time value of 200 was used instead of 25 in order to show more detail in both sets. The **`M-Set`** figure is shown after 3 zoom clicks in the seahorse valley.[60] Be patient when increasing the escape time value. The beauty of the final fractal image is usually worth the wait! In the **`M-Set`** figure, a cross-hair is positioned in an area of the M-Set known as the upper seahorse valley. The corresponding Julia Set for this location is displayed in the **`J-Set`** figure. You may not be able to exactly hit the point shown in the **`M-Set`** figure, but you should be able to get fairly close. Even so, your Julia Set may be a bit different than the one displayed here.



`Figure M-Set after 3 zooms.  Note the cross!`

The Julia Set on the next page corresponds to the complex seed point (-0.753333333333, 0.0316666666667i). You can now zoom in on the Julia Set by clicking on any region or point of interest. To return to the M-Set, simply press the "**`m`**" key.

The beauty and uniqueness of the M-Set (and the subsequent Julia Sets!) are striking to behold. There have been (and continue to be) an amazing number of papers, articles, and texts based on explorations and research about the M-Set. Not only can you find interesting shapes in the M-Set, but we can make associations with prime numbers and PI. We are now going to work with some exercises based on the programs we've written in this section. Make certain that you save any modifications to

[59] Actually EACH POINT in the M-Set corresponds to a different Julia Set. Since points have no dimension, there are an infinite number of Julia Sets possible from the M-Set!
[60] Believe it or not, each area of the Mandelbrot Set has been named after the shapes of the "whorls and swirls" inside the set in that region.

the programs (particularly the last **pymandeljulia.py** program!) under a new filename to preserve the original program listings. Oh, by the way… do you see the Cardiod in the M-Set? Hmmm….

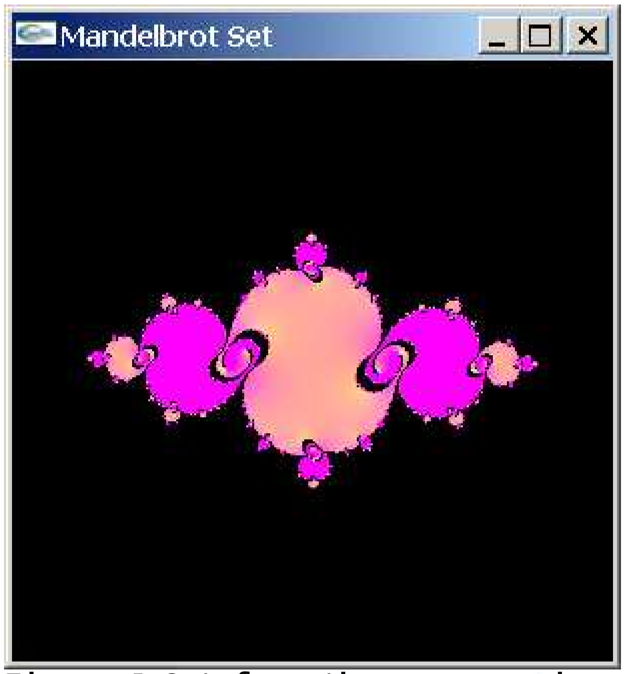


Figure J-Set from the cross region in the M-Set above.

Exercises

1) The coloration of the M-Set is one of the aspects that make the exploration of the M-Set so striking. Experiment with the **glColor** statements and see what you can accomplish.

2) Details of the M-Set can be amazing to view. In order to provide finer details, we need to increase the value of the escape time. Find this code section in the listing (it is commented) and increase the value. See if you can find an optimum value for both speed of execution and detail.

3) Zoom, zoom, zoom! Choose various regions of the M-Set and simply zoom. How far can you zoom before screen glitches appear? Can you fix those glitches? How might you include something in the code so that you would know the level of magnification? Where would you print this magnification level? One of the interesting things about the M-Set is that after a few zooms, it is quite possible that you are looking at something never before seen by human eyes. That makes you an explorer! Where can you change the zoom factor (both in and out) so that you can increase the zoom increment more quickly? Figure **Exercise 3** illustrates a 9 "click" zoom into the seahorse valley region. You may need to change coloration or increase the escape time to get an optimally detailed plot. The resulting plot is usually worth the increased wait time!

4) Can you find some "mini" M-Sets around the main M-Set? Where are these located? Are they identical to the main M-Set? How are they alike and different? Figure **`Exercise 4`** shows one such "mini" M-Set. Can you find where this one is located?

5) Explore the M-Set and Julia Set connections. Is there a difference in appearance between Julia Sets within the M-Set, on the border of the M-Set, and external to the M-Set? Can you use the appearance of the M-Set in a particular region to predict the appearance of the Julia Set?

6) Explore different equations for the M-Set such as:

a)	**`z = z**3 + a`**	
b)	**`z = exp(z) + a`**	(escape distance **`zz < 50`**)
c)	**`z = sin(z) + a`**	(escape distance **`zz < 50`**)
d)	**`z = cos(z) + a`**	(escape distance **`zz < 50`**)
e)	**`z = z**4 + a`**	

You may need to increase the escape distance value (commented in the code) to achieve a decent plot. You might have to change **`axrng`** to a different (probably larger) value to entirely encompass the plot. Make up your own equations and see what happens! The equations above are shown in figures **`Exercise 6a`** through **`Exercise 6e`**. Zoom in and out to achieve interesting results. With the **`exp()`** and trig functions, plotting time is increased, so have some patience. You can decrease the escape time value (commented in the code) to increase the speed of the plot at the sacrifice of detail. Sometimes, though, coloration is more important than detail so experiment there as well. ** Try to plot some Julia Sets in and around these M-Sets! Amazing!

7) Can you figure out how to add a timing feature to the M-Set program to calculate how long it takes to plot the M-Set? If not, look at the next exercise for a possible solution.

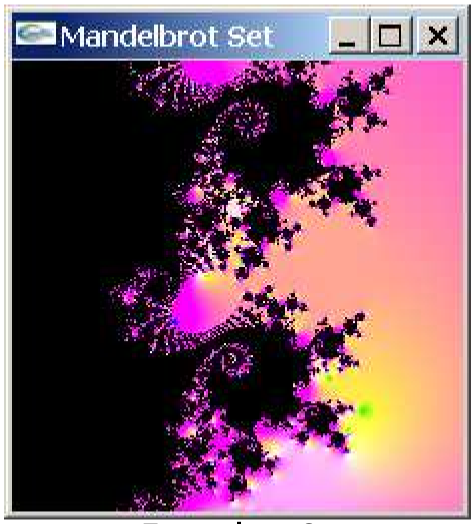
8) Unfortunately there is no inverse iteration method for the M-Set. However, we can use a method that scans for the boundary of the M-Set and only plots those points that are on the boundary. This method quickly draws the outline of the M-Set. The following listing is an implementation of the boundary scanning method.

```
# PyBoundMSet.py
# Plot an M-Set
# Using boundary scanning

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from random import *
from Numeric import *
from time import *
import sys

# If psyco isn't installed, delete the next two lines!
```

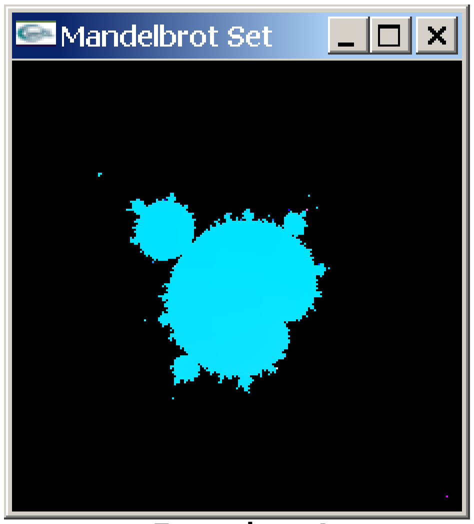
```
import psyco
psyco.full()

axrng = 2.0
width = 400
height = 400
hstep = 2.0*axrng/width
vstep = 2.0*axrng/height

def init():
   glClearColor(0.0, 0.0, 0.0, 0.0)
   gluOrtho2D(-axrng,axrng,-axrng,axrng)

def drawboundmandel():
   global escape
   glClear(GL_COLOR_BUFFER_BIT)

   # for a timer!
   tim = time()

   # Chooses the number of random pixels to check
   # Increase this number for a more dense plot.
   for i in range(0,100000):

              # limits the M-Set ranges
              # to speed up execution
              # and choose random pixels in
              # this more limited range
              x = -2*axrng*random()+.5
              y = 1.25*axrng*random()-1.25

              # draws a triangle around the M-Set
              # So we have fewer points to choose from
              if y < .625*x + 1.25 and y > -.625*x - 1.25:

                   bound = 0

                   # check pixels at North, South,
                   # East, and West locations to see
                   # if these points escape to infinity
                   # If so, add 1 to bound variable
                   leng = escapetime(x+hstep,y)
                   if leng < 2:
                        bound += 1
                   leng = escapetime(x-hstep,y)
                   if leng < 2:
                        bound += 1
                   leng = escapetime(x,y-vstep)
                   if leng < 2:
                        bound += 1
                   leng = escapetime(x,y+vstep)
                   if leng < 2:
                         bound += 1
                   glBegin(GL_POINTS)

                   # If any, but not ALL neighboring
                   # pixels escape, then the current pixel
                   # is on the border of the M-Set so plot it!
                   #
                   # Change second bound to bound < 5 for
                   # an interesting effect!
                   if bound > 0 and bound < 4:
                         glColor3ub(85*bound,50*bound,90*bound)
                         glVertex2f(x,y)
                   glEnd()
                   glFlush()

   # Print the elapsed time in the console window
   print time() - tim

def escapetime(x,y):
   n = 0
   z = a = complex(x,y)

   # Low escape time for a quick plot
   while n < 25:

        # M-Set equation
        z = z**2 + a
        zz = abs(z)

        # escape distance
        if zz > 2:
              n = 5001
        n += 1
   return zz

def main():
   glutInit(sys.argv)
   glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE)
   glutInitWindowPosition(50, 50)
   glutInitWindowSize(width, height)
   glutCreateWindow("Julia Set")
   glutDisplayFunc(drawboundmandel)
   init()
   glutMainLoop()

main()

# End Program
```

The resulting plot for this program is shown in figure **Exercise 8a**. Notice how quickly we see an outline of the M-Set! As we did with the normal escape time M-

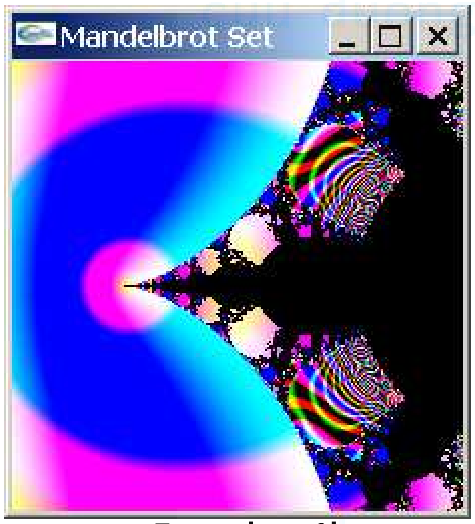
Set program, try experimenting with escape time and try different M-Set equations to see what happens.[61] Also, increase the number of pixels to plot (commented in the program) and change the second bound conditional variable to a value greater than 4 for an interesting effect (also commented). Figure **`Exercise 8b`** illustrates an escape time of 250. The M-Set plot is more detailed, but takes longer to display.



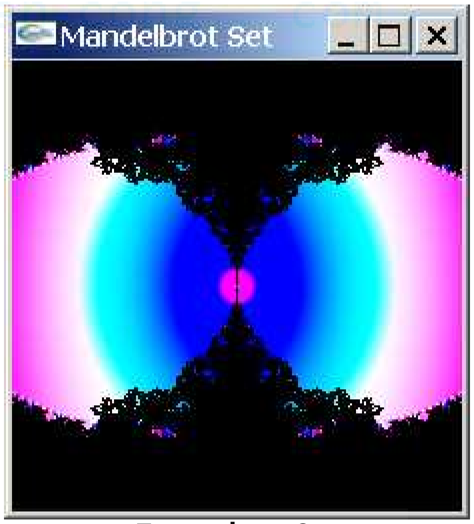
`Exercise 3`



`Exercise 4`

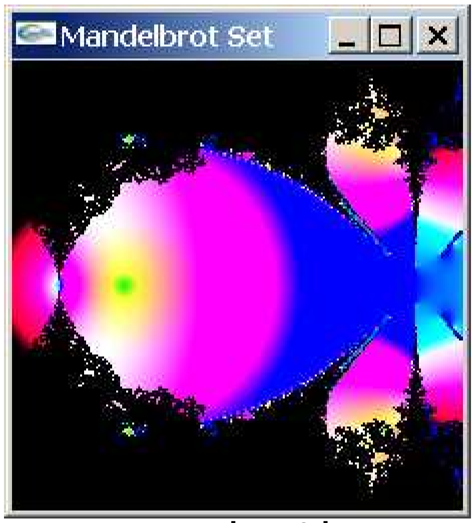


`Exercise 6b`

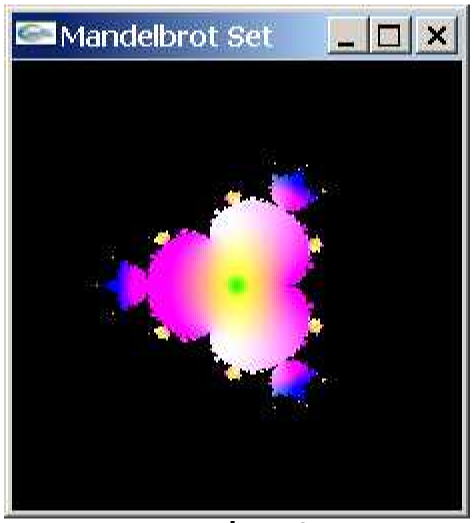


`Exercise 6c`

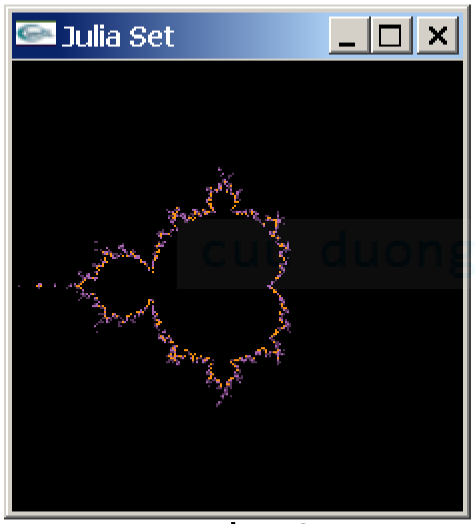
[61] You will have to alter the x and y ranges at the beginning of the plotting loop to see the complete plots of other equations. These ranges are for the M-Set only. You can experiment to see what works in each new equation.



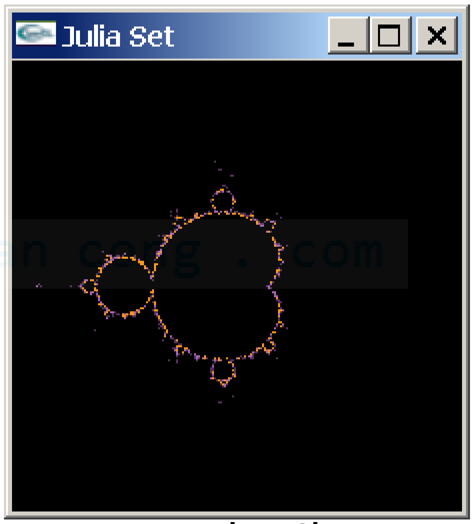
Exercise 6d



Exercise 6e



Exercise 8a



Exercise 8b

Chapter 8   2D Animation

How do we make something move? Perhaps before we ask that question, we should ask "How do we know when an object is moving?" In order to determine whether or not motion is occurring we must first establish a reference point that we define to be stationary. Then, any difference in the relative positions of the object in question and the reference point would signify object movement or motion. The earth serves as an excellent reference point for everyday situations. We assume that the earth is stationary, so any change in the position of an object on the earth is easily interpreted as motion by that object. Sometimes if we move the "stationary" reference point and keep the object in place, we can trick our brain into believing that an object is in motion. For example, in cartoon animation we can hold an object such as an airplane stationary with respect to the drawing frame, but at the same time move the background scenery from right to left. This gives the appearance of an airplane flying from left to right! Many older computer games involved this form of background movement to simulate the motion of a spaceship or aircraft. Another form of cartoon style animation involves drawing an object in a particular location, then in the next frame, erasing and redrawing the object in a slightly different location. If we do this drawing, erasing, and redrawing sequence enough times at a relatively high speed, we can create the illusion or appearance of motion. We will use this method to approach animation in Python.

Section 8.1   Follow the Bouncing Ball

For our first example of animation we are going to have a sphere or ball bounce around the confines of a graphics window. This task is not as simple as it sounds. First, we have to draw a ball... that's not too difficult. Then we have to erase the ball and draw it in a different location. Again, not an insurmountable challenge. Finally, we have to figure out how to have the ball realize that it has hit the side of the graphics window so that it can bounce off the "wall" and reverse its movement. Collision detection can be somewhat tricky! Not only must we detect when the ball has collided with the wall, we must also make certain that its rebound is realistic. In other words, we must make every attempt to insure that the ball follows the laws of physics. Without further ado, here is a first attempt at animation.

```
# PyBounce.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys

# uncomment these lines later
# to see if there is any difference
# in the speed of the ball
# import psyco
# psyco.full()

#  globals for animation, ball position
#  and direction of motion
global anim, x, y ,dx, dy
```

```
# initial position of the ball
x = -0.67
y = 0.34

# Direction "sign" of the ball's motion
dx = dy = 1

# Window dimensions
width = height = 500
axrng = 1.0

# No animation to start
anim = 0

def init():
     glClearColor(0.0, 0.0, 0.0, 1.0)
     glColor3ub(255, 0, 0)

     # Dimensions of the screen
     # Make axrng larger and see what happens!
     gluOrtho2D(-axrng, axrng, -axrng, axrng)

def idle():
     # We animate only if anim == 1, otherwise
     # the ball doesn't move
     if anim == 1:
          glutPostRedisplay()

def bounce():
     global x, y, dx, dy
     glClear(GL_COLOR_BUFFER_BIT)

     # changes x and y
     x += 0.001*dx
     y += 0.001*dy

     # Keep the motion mathematics
     # Safe from harm and then
     # Move the ball location based on x and y
     glPushMatrix()
     glTranslate(x,y,0)
     glutSolidSphere(0.1, 50, 50)
     glPopMatrix()

     # Collision detection!
     # What happens here and why does this work?
     if x >= axrng or x <= -axrng:
          dx = -1*dx
     if y >= axrng or y <= -axrng:
          dy = -1*dy
```

```
	glFlush()

def keyboard(key, x, y):
	#  Allows us to quit by pressing 'Esc' or 'q'
	#  We can animate by "a" and stop by "s"
	global anim
	if key == chr(27):
		sys.exit()
	if key == "a":
		# Notice we are making anim = 1
		# What does this mean?  Look at the idle function
		anim = 1
	if key == "s":
		# STOP the ball!
		anim = 0
	if key == "q":
		sys.exit()

def main():
	glutInit(sys.argv)
	glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
	glutInitWindowPosition(100,100)
	glutInitWindowSize(width,height)
	glutCreateWindow("PyBounce")
	glutDisplayFunc(bounce)
	glutKeyboardFunc(keyboard)
	glutIdleFunc(idle)

	init()
	glutMainLoop()

main()

# End of program
```

When you run this program, you should first see a window that looks like **Figure 8.1** on the next page.   Press "**a**" and see what happens. You should see the ball bouncing around the screen, but it doesn't look very smooth. You may even notice that the ball is sinking into the sides of the screen prior to rebounding. We will fix these problems in a moment, but for now let's look at the code a bit more closely.

```
# PyBounce.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys

# uncomment these lines later
# to see if there is any difference
# in the speed of the ball
# import psyco
# psyco.full()
```

There is nothing mysterious about the beginning lines of the program. We've seen this code before. We don't need the **Numeric** module at this point, so we do not import it. Once the program is running properly, you may uncomment the psyco code to see if your program runs a bit more smoothly. However, you may want to wait until we fix the animation issues first.

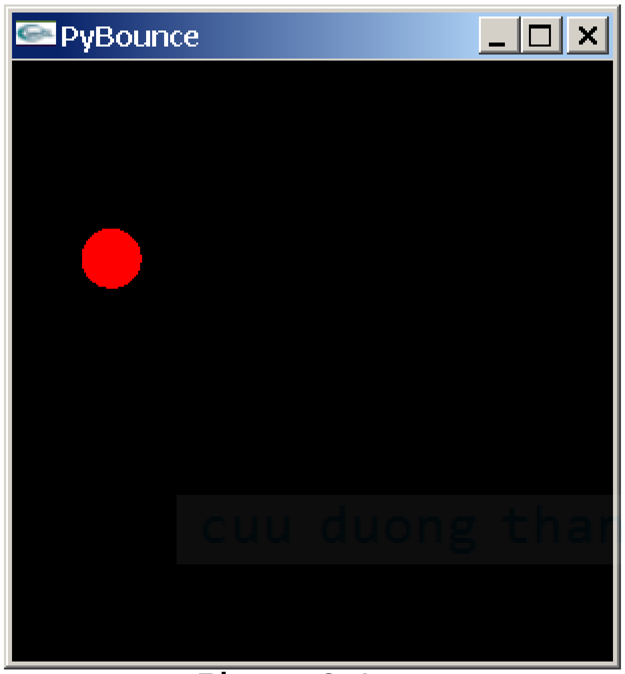


Figure 8.1

The next section declares some global variables and sets some initial values for those variables:

```
#  globals for animation, ball position
#  and direction of motion
global anim, x, y ,dx, dy

# initial position of the ball
x = -0.67
y = 0.34

# Direction "sign" of the ball's motion
dx = dy = 1

# Window dimensions
width = height = 500
axrng = 1.0

# No animation to start
anim = 0
```

The **global** declaration statement looks a bit different, doesn't it? We can declare all **global** variables using one statement if we wish. This single line declaration has not been used in previous programs because my goal was clarity rather than efficiency. We have advanced to the point now where we can begin to take a few shortcuts as long as we retain the meaning of the code construction.

The **x** and **y** variables provide the initial position of the center of the ball. The direction variables, **dx** and **dy**, are both initialized to +1. We will use these two variables to establish the direction that the ball is traveling. When the ball collides with a wall, changing the sign of **dx** or **dy** will allow us to change the direction of the ball's motion as we shall soon see. There is nothing new about the **width** and **height** variables other than the format of how we are assigning their initial values. The global variable **anim** will be used to toggle the animation of the ball on or off by using the "**a**" key.

The **def init():** function is nearly identical to earlier programs, but following the **init** function, we see a new function called **def idle():**. The **def idle():** function is called when GLUT is "idling" or not responding to mouse or keyboard input. Similar to the GLUT **mouse** and **keyboard** functions, we must declare the **def idle():** function in the **def main():** code block by using **glutIdleFunc(idle)**. Of course, we could name this function anything we wish as long as we declared the identical name in the **glutIdleFunc()** statement. However, using the name "**idle**" seems to make sense for our first program. It is the **idle** function that allows animation to occur. As long as GLUT is NOT responding to keystrokes or mouse input, it "idles" for us by executing the following code:

```
def idle():
      # We animate only if anim == 1, otherwise
      # the ball doesn't move
      if anim == 1:
           glutPostRedisplay()
```

In this code block, **IF** the global variable **anim == 1**, then Python simply keeps calling the display function by issuing a **glutPostRedisplay()** command. This continuous calling of the display function may not seem important, but the result in our current program is animation! If **anim <> 1**, then animation stops. Of course, the actual animation procedures occurs in the display function **def bounce():**

```
# display function

def bounce():
      global x, y, dx, dy
      glClear(GL_COLOR_BUFFER_BIT)

      # changes x and y coordinates
      x += 0.001*dx
      y += 0.001*dy

      # Keep the origin in safe keeping
      # for glTranslate!
```

```
        glPushMatrix()

        # Move the ball location based on x and y
        glTranslate(x,y,0)
        glutSolidSphere(0.1, 50, 50)
        glPopMatrix()

        # Collision detection!
        # What happens here and why does this work?
        if x >= axrng or x <= -axrng:
              dx = -1*dx
        if y >= axrng or y <= -axrng:
              dy = -1*dy

        glFlush()
```

It is important to understand the purpose of the **glClear** statement. Comment this statement and run the program (make sure you press "**a**" to start the animation). Uncomment **glClear** after you have seen the results. What purpose does the **glClear** function serve? That's correct! **glClear** allows for the erasing and subsequent redrawing needed for proper animation. The following statements:

```
        x += 0.001*dx
        y += 0.001*dy
```

continuously increment the **x** and **y** coordinates by **0.001**.[1] The direction or sign of the **0.001** value in each equation is determined by the value of **dx** and **dy** respectively.[2] Initially, **dx** and **dy** are both **+1** (how do you know?) and the **x** and **y** coordinates increase by **+0.001** each iteration. This change in **x** and **y** coordinates along with the erasing and redrawing of the ball results in the appearance of motion. If we change **dx** and/or **dy** to a value of **-1**, we will change the horizontal and vertical directions of the **x** and **y** motions by subtracting **0.001** from **x** and/or **y**. So, the values of **dx** and **dy** together determine which direction the ball will move in the graphics display window.

The next section is definitely mysterious (preceding comments have been removed):

```
        glPushMatrix()
        glTranslate(x,y,0)
        glutSolidSphere(0.1, 50, 50)
        glPopMatrix()
```

[1] Remember what **+=** means! It is equivalent to **x = x + 0.001*dx**.

[2] When we write a program to simulate the dynamics of star movements, we will call such direction routines unit vectors. A vector has both magnitude and direction. In this case, we do not want to change the magnitude of the motion using **dx** and **dy**, only the direction of motion. Hence, we multiply **0.001** by **-1** or **+1**, which changes the direction sign, but not the absolute value of the velocity of the ball..

What is **`glPushMatrix()`** and why do I have to use it? First, you MUST remember that for every **`glPushMatrix()`**, you must have a matching **`glPopMatrix()`**. Every code statement between these two matrix commands is "protected" from inadvertent changes in the screen origin or reference point. What do I mean by inadvertent changes in the current graphics reference point? The **`glTranslate(x,y,0)`**[3] statement allows us to move to any **`x,y`** coordinate location in the graphics window (and beyond!).[4] However, issuing a simple unprotected **`glTranslate()`** command also changes the "origin" or the reference point of the **`x,y`** coordinates within the **`glTranslate`** statement. Every future point we select will be relative to the last point we specified in **`glTranslate`**. We don't want that behavior in our program! Here is an example: Say I issue an initial **`glTranslate(0.5, 0.5, 0)`** command and then draw a circle. As expected, the circle will be centered at (0.5, 0.5) on the screen. If I issue the same **`glTranslate`** command again, you might expect that the circle will not move... after all, we are using the same coordinates of (0.5, 0.5) and we should draw a new circle in the same location. In actuality, though, a new circle will be drawn (0.5, 0.5) units away from the previous circle! If we continue to issue the same **`glTranslate`** command, we will create a line of individual circles on the **`y = x`** diagonal. Even though we issue the SAME **`glTranslate`** command, each new circle will be plotted relative to the previous circle. While the effect may be interesting, this line of identical circles is probably not what we want!

So, effectively **`glTranslate`** will create a NEW origin or reference point every time we issue the command. This is NOT what we want. We want the origin to stay in the CENTER of our screen and have the coordinates (0, 0) like we and Descartes expect.[5] We want to be able to use **`glTranslate`** and plot individual circles wherever we like using the traditional **`x`** and **`y`** coordinate system. We do NOT want our circles to march merrily off into the void! This is the reason we use **`glPushMatrix()`** and **`glPopMatrix()`**. Using these commands as bookends to our plotting statements, we "freeze" the origin in the correct location and we are able to use **`glTranslate`** exactly as expected. If you don't quite understand this concept, simply comment out both **`glPushMatrix()`** and **`glPopMatrix()`** statements and run the program again. What happens when you press "a". See what I mean? The ball takes off on a diagonal and never returns. You should uncomment the "matrix" statements before someone gets hurt!

In the process of explaining the **`glPushMatrix()`** and **`glPopMatrix()`** statements, I couldn't avoid discussing **`glTranslate`**, so hopefully you understand the purpose of this command. A translation is a lateral movement in a plane or in space, hence the name **`glTranslate`**. We used **`glVertex2f`** (and later we will use **`glVertex3f`**) to plot points ONLY. If we want to position objects other than points (such as spheres), we must use **`glTranslate`** to move the object to the desired location. Finally, the line after **`glTranslate`** plots a sphere using the appropriate GLUT command. The format of this command is as follows:

[3] So, what does the "**`0`**" in **`glTranslate(x,y,0)`** do? OpenGL is a 3D graphics environment by nature, so the **`0`** holds a place for the z-axis, which we'll be using later on.
[4] In geometry, a translation is a movement from one (x, y) location to another (x', y') location in a linear direction. I use the phrase 'lateral movement' in the text.
[5] Ah, yes... look up René Descartes and see what he did for algebra and geometry!

```
glutSolidSphere(radius, slices, stacks)
```

The meaning of **`radius`** is obvious and its value should be chosen to fit the axis range of your graphics window. If you have set **`axrng = 1.0`**, then you should choose a radius much smaller than this value or the sphere (ball) will be much too large. The **`slices`** and **`stacks`** parameters are analogous to longitude and latitude lines. The higher these values, the more smooth the sphere, but it will take longer to draw. If your animation is much too slow or "jerky", then choose smaller values for **`slices`** and **`stacks`**. I used 50 for both values, but you can get a pleasing sphere/ball using 10 or 20 for **`slices`** and **`stacks`**. One question you might pose is "Why are you plotting a sphere when we are using 2D graphics?" Good question. The short answer is that the cross-section of a sphere is a circle, which looks like a "ball" on the graphics screen and serves our purpose nicely. GLUT spheres are very easy to plot!

The next section of code is for simple collision detection.[6] Without this code, the ball will move off the screen (similar to what happened when we commented the **`glPushMatrix()`** and **`glPopMatrix()`** commands) never to return. Here are the magic collision detection statements:

```
if x >= axrng or x <= -axrng:
     dx = -1*dx
if y >= axrng or y <= -axrng:
     dy = -1*dy
```

That is all it takes! If the **`x`** location of our ball is at or outside the +/- limits of **`axrng`**, then we multiply **`dx`** by -1, effectively changing the sign of **`dx`** and hence the direction sign of **`0.001`**! The same reasoning may be used for the **`y`** location of the ball. Essentially, if the ball reaches any screen boundary as set by **`axrng`**, then the ball will reverse its direction. This is a rudimentary, yet effective form of collision detection. Look once again at:

```
x += 0.001*dx
y += 0.001*dy
```

to see how the signs of **`dx`** and **`dy`** affect the values of **`0.001`**, **`x,`** and **`y`**, and thus the motion of the ball. Finally, we use **`glFlush()`** (perhaps for the last time...) to draw the new location of the ball.

The **`keyboard`** function is a bit more complex than in previous programs because we are adding two new key options:

```
def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'
     #  We can animate by "a" and stop by "s"
     global anim
     if key == chr(27):
```

[6] Collision detection can become not so simple for complex objects. Collision response can be even more complicated if we want the physics to be accurate!

```
          sys.exit()
     if key == "a":
          # Notice we are making anim = 1
          # What does this mean?  Look at the idle function
          anim = 1
     if key == "s":
          # STOP the ball!
          anim = 0
     if key == "q":
          sys.exit()
```

In addition to the usual "**Esc**" and "**q**" keys, we are allowing the user to start animation by pressing the "**a**" key (how does this start animation?) and to stop animation by pressing the "**s**" key (again, how does this stop animation?). Note the **global anim** statement. This statement allows us to change the value of **anim** within the **keyboard** function so the entire program can use the new value of **anim**.

The only modification to the **def main():** function is the addition of the **glutIdleFunc(idle)** statement, which is needed in order to define the GLUT **idle** function.

The animation provided by this particular program has an annoying flicker and is not pleasing to the eye. Let's fix that visual problem. First, we must deal with the flickering effect. Ideally, we would want to see much smoother animation. Fortunately, the "fix" is simple. Make the following changes to your program listing:
In **def main():** replace:

```
     glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE)
```

with:

```
     glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE)
```

and in **def bounce():** replace our good friend:

```
     glFlush()
```

with:

```
     glutSwapBuffers()
```

Remember to keep the same level of indentation when you make the modifications above and then run the program again. See how much smoother the animation is! This is called "double-buffering" and the concept is rather simple. Imagine two graphics windows (called "buffers"), one in front of the other. While we draw on the back window, we display the front window. We then "swap buffers" and exchange the windows, again drawing on the "hidden" window/buffer while displaying the new recently drawn buffer. We continue to repeat this process and the result is smooth animation. This technique is much more efficient than the original drawing-erasing-drawing strategy, which caused flickering.

The second issue is a bit more difficult to treat. The bouncing ball is sinking into the walls of the screen. We don't want that behavior, so what do we do? Before we can fix the problem, we must understand why the ball behaves as it does. Remember that the position of the ball as set by **`glTranslate`** is based on the center of the ball. So, while we want the ball to change direction based on contact with its outer border and the wall, it actually changes its direction when the center point hits a wall. The fix is not difficult, though, and is illustrated by the following statements:

```
if x >= axrng - 0.1 or x <= -axrng + 0.1:
     dx = -1*dx
if y >= axrng - 0.1 or y <= -axrng + 0.1:
     dy = -1*dy
```

What is the significance of the **`-0.1`** and **`+0.1`** modifications? These represent the radius of the ball! By using these values, we take into consideration the radial size of the ball and can detect a window border collision with the outside boundary of the ball. If you haven't done so, run the program and watch what happens. This is much better animation than the original **pybounce.py** program and we've not had to expend much additional effort. You should understand that it would be more efficient to use a variable to represent the radius of the ball/sphere everywhere in the program that the radius is needed. How might you do that? See Exercise 2!

In the following exercises, be prepared to invent, experiment, and discover additional features and behaviors dealing with simple animation.

Exercises

1) Try animating the different GLUT shapes listed below! Does the collision mechanism still work properly for each?

```
glutSolidCone(base, height, slices, stacks)
glutSolidCube(size)
glutSoliddodecahedron()
glutSolidIcosahedron()
glutSolidOctahedron()
glutSolidTetrahedron()
glutSolidTorus(inner_radius, outer_radius, sides, rings)
glutSolidTeapot(radius)
```

Can you fix the collision mechanism if it is not working? How? Some of the GLUT shapes take no parameters (which ones?) so how can you set their size? There is an OpenGL function, **`glScalef(x, y, z)`**, which will allow you to change the size of the object you are drawing. If you place **`glScalef(2.0, 2.0, 2.0)`** immediately above the GLUT shape command, then you will be multiplying the size/scale of the shape by 2.0 on all three axes, (x, y, z). The effect would be a doubling of the linear dimensions of the shape. Likewise, placing **`glScalef(0.5, 0.5, 0.5)`** prior to issuing a GLUT shape command will scale each of the axes by 0.5, effectively shrinking the image. What happens if you use unequal values for the axes? Try it! Also, you can substitute the word "**`Wire`**" for "**`Solid`**" in each of the GLUT shape commands with the expected result.

2) We have "hard-coded" the radius of the sphere/ball in the example program by using 0.1 in **`glutSolidSphere`** and in the collision detection code. Replace this 0.1 value with an appropriate variable so that when you wish to change the radius of the sphere, you can do so in only one place and the change will be reflected throughout the program. Can you say "**`global`**"?

3) Try different initial values for the position and velocities of the bouncing ball. Also, try increasing **`axrng`** to a much larger number. What happens to the apparent size of the ball? Why does this happen? Do you notice any other effects as the size of **`axrng`** increases? Does the ball ever become a ring? Why do you think this happens?

4) If you change the dimension of the graphics window while the program is running by dragging the lower right corner or by maximizing the window, what happens to the bouncing ball? What are we missing in this program that might fix this problem? If you thought "a reshape function", then give yourself a pat on the back. However, simply using the **`reshape`** function from previous programs will not suffice. We must take into consideration the graphics axis ranges for collision detection. Here is an example of the steps needed to add a **`reshape`** function to the **pybounce.py** program:

First add:

```
global xborder, yborder
```

to the **`global`** variable section. We will use these two variables, **`xborder`** and **`yborder`**, to establish virtual "walls" within the graphics window. Then initialize these variables to the value for **`axrng`** as follows:

```
xborder = yborder = axrng
```

Why must you place this line AFTER you set the value for **`axrng`**? Next, add the following **`def reshape(w, h):`** function:

```
def reshape(  w,  h):
   global xborder, yborder
   # To insure we don't have a zero height
   if h==0:
        h = 1

   #  Fill the entire graphics window!
   glViewport(0, 0, w, h)

   #  Set the projection matrix... our "view"
   glMatrixMode(GL_PROJECTION)
   glLoadIdentity()
   #  Set the aspect ratio of the plot so that it
   #  Always looks "OK" and never distorted.
   if w <= h:
        gluOrtho2D(-axrng, axrng, -axrng*h/w, axrng*h/w)
        yborder = axrng*h/w
        xborder = axrng
   else:
        gluOrtho2D(-axrng*w/h, axrng*w/h, -axrng, axrng)
        xborder = axrng*w/h
        yborder = axrng

   #  Set the matrix for the object we are drawing
   glMatrixMode(GL_MODELVIEW)
   glLoadIdentity()

# End Function
```

Note the:

```
   yborder = axrng*h/w
   xborder = axrng
```

and

```
   xborder = axrng*w/h
   yborder = axrng
```

statements in the **`if w <=h:... else:`** conditional block of code. These statements will establish the new "reshaped" boundaries for the walls of the graphics window based on **`axrng`** and the aspect ratio (**`h/w`** or **`w/h`**) of the reshaped window. What must you do in **`def main():`** to let Python know that this **`reshape`** function exists? That's correct, you must add a **`glutReshapeFunc(reshape)`** statement.

Finally, we must make some minor changes to the collision conditional statements in the **`def bounce():`** display function:

```
   if x >= xborder-radius or x <= -xborder+radius:
        dx = -1*dx
   if y >= yborder-radius or y <= -yborder+radius:
        dy = -1*dy
```

We are using the new **`xborder`** and **`yborder`** global variables to detect wall collisions. You DID implement a **`radius`** variable for the ball in exercise 2, didn't you?

Perhaps a good question to ask would be "Why would I want to change the window dimensions while the program is running?" Good question... however, try resizing the window and watch how the path of the bouncing ball changes.

One caveat: If you resize the window more than once, it is possible to "trap" the ball outside the graphics windows boundaries and the ball will never reappear. Try it! See if you can "break" the simulation by making the ball disappear. How might we fix this problem? Once solution would be to detect the location of the ball using the **x**,**y** coordinates and place the ball "back into play". Type the following code immediately after the **if w<=h:** block (and indented at the same level as the **if w<=h:** statement!) within the **def reshape** function:

```
if x <= -xborder:
     x = -xborder + (2*radius)
if x >= xborder:
     x = xborder - (2*radius)
if y <= -yborder:
     y = -yborder + (2*radius)
if y >= yborder:
     y = yborder - (2*radius)
```

You will also have to add **x** and **y** to the global variable statement in the first line of the **reshape** function. This **if** code block is fairly crude, but it seems to work. Basically the code tests the position of the ball's (x, y) coordinates against the **+/- xborder** and **+/- yborder** graphics window "walls". If the ball is outside those boundaries, then it resets the position of the ball within the graphics window by an arbitrary value of **2*radius** or a single ball diameter. Experiment with this code by resizing the graphics window during animation and see what happens.

5) How would we add some interactive capability to this bouncing ball simulation? In other words, how can we control the motion of the ball by using key presses? We already know how to implement a key press to end a program and to start and stop animation in this simulation. Let's learn something new and add the arrow or cursor keys to our keyboard interaction choices. In order to accomplish this task, we can not simply use the current **keyboard** function. The arrow or cursor keys are a bit different and we must access them through a "special" key function. Try the following changes in your **pybounce.py** program:

Add the following new function to your code listing, remembering to have NO indentation for the **def** statement:

```
def specialkey(key , x, y):
   global hvel, vvel
   if key == GLUT_KEY_LEFT:
        hvel -= 0.001
   if key == GLUT_KEY_RIGHT:
        hvel += 0.001
   if key == GLUT_KEY_UP:
        vvel += 0.001
   if key == GLUT_KEY_DOWN:
        vvel -= 0.001

# End specialkey function
```

You probably can gather from the structure of this code that we are going to use the **LEFT**, **RIGHT**, **UP**, and **DOWN** keys depending on the value stored in the variable **key**. We will need to register this function with Python and OpenGL/GLUT by adding the following line to **def main():** immediately above the **idle()** statement.

```
glutSpecialFunc(specialkey)
```

We need to change the initial global variable statements as follows:

```
global width, height, axrng, anim, x, y
global xborder, yborder, radius, hvel, vvel
```

Note the deletion of the dx and dy variables and the addition of our old friends (from the physics **pycannon.py** program) **hvel** and **vvel**. Now add:

```
hvel = vvel = 0.000
```

to the variable initialization section of your program following the global variable statements. Three changes need to be made in the **def bounce():** display function. First, change the global variable statement to:

```
global x, y, hvel, vvel
```

and then modify the position/motion equations as follows:

```
x = x + hvel
y = y + vvel
```

Since we are going to control the motion of the ball using the arrow keys, we no longer need to designate a constant speed (which used to be **0.005**) nor do we need **dx** and **dy** to change directions after a wall collision. We will use **hvel** and **vvel** for both purposes. Finally, change the collision detection statements to:

```
if x >= xborder-radius or x <= -xborder+radius:
     hvel = -1*hvel
if y >= yborder-radius or y <= -yborder+radius:
     vvel = -1*vvel
```

Notice the addition/substitution of the **hvel** and **vvel** variables in place of **dx** and **dy**. At this point you should be able to run your program. Press the '**a**' key to start the animation and then see if you can move the ball using the arrow keys. Can you add an additional key press that will set both **hvel** and **vvel** to zero, effectively stopping the ball? You might think that pressing the '**s**' key will serve this purpose. It is true that animation ceases with the '**s**' key, but as soon as the '**a**' key is pressed, the ball will resume the motion it had prior to stopping. Not only that, but you can change the stored values for **hvel** and **vvel** while the ball is in a state of suspended animation! What we need is a method of actually bringing the velocity of the ball to zero or to the rest state. See if you can figure out how to do bring the velocity of the ball to zero on your own.

6) As a final exercise in this section, can you add friction to this simulation so that the motion of the ball will eventually come to a halt without additional force (arrow key presses) being added? See the **pycannon.py** program in Section 5.4, Exercise 12 for a hint.

Section 8.2   A Little Gravity!

In this section, let's expand the model that we created in the last program and add some gravity to the simulation. What we want to observe is a ball acting under the influence of a gravitational field, detecting collisions with the walls, floor, and possibly ceiling, and unaffected by friction at this point. The following is a complete code listing of such a simulation. You may simply load your last program from the previous section and make changes where appropriate. Save this modified program as **pygravity.py** or something similar.

```
# PyGravity.py

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys

# uncomment these lines later
# to see if there is any difference
# in the speed of the ball
# import psyco
# psyco.full()

#  globals for animation, ball position
#  and direction of motion
global anim, x, y ,hvel, vvel, radius
global xborder, yborder

# initial position of the ball
x = -0.67
y = 0.34
dtime = 0.0005
radius = 0.1
hvel = 0.75
vvel = 3.0

# Window dimensions
width = height = 600
xborder = yborder = axrng = 1.0

# No animation to start
anim = 0

def init():
     glClearColor(0.0, 0.0, 0.0, 1.0)
     glColor3ub(255, 0, 0)

     # Dimensions of the screen
     # Make axrng larger and see what happens!
     gluOrtho2D(-axrng, axrng, -axrng, axrng)
```

```
def idle():
     # We animate only if anim == 1, otherwise
     # the ball doesn't move
     if anim == 1:
          glutPostRedisplay()

def plotfunc():
     global x, y, hvel, vvel
     glClear(GL_COLOR_BUFFER_BIT)

     # changes x and y
     x += hvel*dtime

     # earth's gravity -9.8 meters per second*second
     vvel = vvel - 9.8*dtime
     y += vvel*dtime

     # This if statement keeps the ball
     # from falling below the window!
     if y <= -axrng + radius:
          y = -axrng + radius

     # Keep the motion mathematics
     # Safe from harm
     glPushMatrix()

     # Move the ball location based on x and y
     glTranslate(x,y,0)
     glutSolidSphere(radius, 50, 50)
     glPopMatrix()

     # Collision detection!
     # What happens here and why does this work?
     if x >= xborder - radius or x <= -xborder + radius:
          hvel = -1*hvel
     if y >= yborder - radius or y <= -yborder + radius:
          vvel = -1*vvel

     glutSwapBuffers()

def reshape(  w,  h):
     global xborder, yborder, x, y
     # To insure we don't have a zero height
     if h==0:
          h = 1

     #  Fill the entire graphics window!
     glViewport(0, 0, w, h)

     #  Set the projection matrix... our "view"
     glMatrixMode(GL_PROJECTION)
```

```
        glLoadIdentity()

        #  Set the aspect ratio of the plot so that it
        #  Always looks "OK" and never distorted.
        if w <= h:
             gluOrtho2D(-axrng, axrng, -axrng*h/w, axrng*h/w)
             yborder = axrng*h/w
             xborder = axrng
        else:
             gluOrtho2D(-axrng*w/h, axrng*w/h, -axrng, axrng)
             xborder = axrng*w/h
             yborder = axrng

        if x <= -xborder:
             x = -xborder + (2*radius)
        if x >= xborder:
             x = xborder - (2*radius)
        if y <= -yborder:
             y = -yborder + (2*radius)
        if y >= yborder:
             y = yborder - (2*radius)

        #  Set the matrix for the object we are drawing
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()

def keyboard(key, x, y):
        #  Allows us to quit by pressing 'Esc' or 'q'
        #  We can animate by "a" and stop by "s"
        global anim
        if key == chr(27):
             sys.exit()
        if key == "a":
             # Notice we are making anim = 1
             # What does this mean?  Look at the idle function
             anim = 1
        if key == "s":
             # STOP the animation!
             anim = 0
        if key == "q":
             sys.exit()

def main():
        glutInit(sys.argv)
        glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE)
        glutInitWindowPosition(100,100)
        glutInitWindowSize(width,height)
        glutCreateWindow("PyBounce")
        glutDisplayFunc(plotfunc)
        glutKeyboardFunc(keyboard)
        glutReshapeFunc(reshape)
        glutIdleFunc(idle)
```

```
    init()
    glutMainLoop()

main()

# End of Program
```

When you have completed the program, run the model and observe what happens... don't forget to press '**a**' to start the animation! Does this animation seem to simulate the motion of a ball under the influence of gravity?

Exercises

1) This is a program that should allow you to "play" with the numbers. Try different values for the initial position of the ball. Try different values for **`dtime`** and see what happens to the simulation. Finally, try different initial velocities (**`hvel`** and **`vvel`**) for the ball and see how the simulation behaves.

2) Try to add a second ball to the simulation and have this second ball act independently of the initial ball.

3) Change the acceleration due to gravity in the following statements:

    ```
    # earth's gravity -9.8 meters per second*second
    vvel = vvel - 9.8*dtime
    y += vvel*dtime
    ```

    In the place of **`9.8`**, you might try **`1.62`**, which represents the moon's gravity. Try **`3.71`**, which is the acceleration due to gravity on the planet Mars. How could you add a global variable to this program to change gravity in the initial section of code? The negative sign in front of the acceleration due to gravity represents an acceleration toward the surface of the planet. A positive sign would represent an acceleration away from the planet's surface. Try changing the sign to **+** and see what happens!

4) As an extension of Exercise 2, can you write some code that will detect a collision between two objects? Detecting a collision and responding properly to a collision are two different things! You might try looking up collision detection and collision response online and get an idea of the difficulties involved.

Section 8.3 A Little MORE Gravity... a 2-Body Simulation

I am pleased to be involved in a project called MSA (Moving Stars Around)[1]. This project is headed by Piet Hut, an astrophysicist and Professor of Interdisciplinary Studies at the Institute for Advanced Studies in Princeton, New Jersey, and Jun Makino, an astrophysicist from the University of Tokyo in Japan. Piet is a friend of mine and I had the distinct honor and privilege of attending a workshop at the Institute in October 2007 to discuss the status and future of MSA. One of the projects we are working on is a simulation of a cluster of stars. In order to simulate a star cluster, one must take into consideration several factors such as the laws of motion, the nuclear reactions within the stars, the interactions of the stars, the gravitational and relativistic effects caused by each star, and other factors which I won't even begin to claim to understand. The complexity of the entire dynamics of even a small cluster of stars is beyond the scope of this text, however I think we can at least glimpse some of the beauty of the dynamics of a star cluster by using the simple laws of motion developed by Newton 300+ years ago. We'll start by looking at a 2 star or 2 body problem.

You'll recall that in Section 7.4 it was mentioned that the 2 body problem is easily solved using Newton's equations, but the 3 body problem (and beyond 3 bodies) is not solvable even with our current knowledge of mathematics and physics. The best we can do is simulate the dynamics of 3 or more bodies using a computer. Well, we have a computer, so what is stopping us...? But before we plunge into a 3 body scenario and beyond, let's start by simulating a simple 2 body system. We will assume that our bodies (stars) are in the vacuum of space and the only forces acting on them are the mutual gravitational attractions between each star. This will be an assumption we will make in future more complex simulations as well. Our equations will be relatively simple. First, we will make use of Newton's law of universal gravitation in equation 1:

$$F = G\frac{m_1 m_2}{r^2} \qquad (1)$$

The law of universal gravitation describes the force of gravity in newtons between any 2 objects if we know the masses of the objects, m_1 and m_2, and the distance, r, between their centers of mass. Big "G" is the the universal gravitational constant first determined experimentally by Henry Cavendish in 1797. It's value is approximately 6.67 x 10^{-11} Nm^2/kg^2. If we know the force acting on an object and we know the mass of the object, we can calculate the acceleration of the object by using Newton's 2nd law of motion in equation 2:

$$F = ma \qquad (2)$$

or if the mass of the object is represented by m_1 equation 2 can be transformed as follows:

[1] See the Art of Computational Science webpage: http://www.artcompsci.org/

$$F = m_1 a \tag{3}$$

Combining Newton's second law with the law of universal gravitation, we can immediately calculate the acceleration of the m_1 object by the following equation:

$$a = G\frac{m_2}{r^2} \tag{4}$$

Since acceleration is a vector, it must have BOTH a magnitude and a direction. The magnitude of the acceleration is calculated from the above equation, but how do we find the direction? Direction is found by multiplying by something called a unit vector. A unit vector has a magnitude equal to 1, so a unit vector can not change the value of a number. However, a unit vector also has a direction which is NOT usually equal to 1, so muliplying by a unit vector WILL change the direction of whatever it multiplies, in this case, acceleration. Here is an example of a unit vector:

$$\frac{\vec{r}}{|r|} \tag{5}$$

Combining the unit vector with equation 4 at the top of this page produces the following formula:

$$a = G\frac{m_2\vec{r}}{r^3} \tag{6}$$

Since I am already making the mathematicians and physicists in the audience cringe by my explanations and equations, let me rewrite equation 6 as follows:

$$a = G\frac{(direction)m_2}{r^3} \tag{7}$$

Equation 7 is provided to illustrate how we will be able to determine the direction of the acceleration vector. Think of the "direction" term as both a distance AND a sign (+/-). The distance is equal to the value of r and will divide out of the denominator, producing the original r^2 term in equation 1. The +/- sign will cross the '=' sign and attach itself to the value of a, the acceleration. Thus, we calculate both a magnitude and a positive or negative direction for acceleration (a). But wait... how can acceleration simply be positive or negative? Can't acceleration be pointed in any direction? Yes, acceleration can be pointed in any direction, but we can break up any vector into x, y, and z (z is used in 3D motion) axis components. When breaking a vector into x, y, and z axis components, only positive and negative directions are needed. This is very handy! Here is what we will be doing (eventually) with multiple body simulations. We will take equation 7 and modify it for each axis component and let r_n stand for the vector (arrow) r in the numerator of equation 6, where n is a coordinate axis:

$$a_x = G\frac{r_x m_2}{r^3} \qquad (8)$$

$$a_y = G\frac{r_y m_2}{r^3} \qquad (9)$$

$$a_z = G\frac{r_z m_2}{r^3} \qquad (10)$$

We have calculated acceleration components in each axis direction in equations 9, 10, and 11. Together, they will combine to describe the acceleration of our star! Now back to our story... Once we know the acceleration of an object, we can calculate its velocity at any time by using similar reasoning:

$$v_x = a_x t \qquad (11)$$

$$v_y = a_y t \qquad (12)$$

$$v_z = a_z t \qquad (13)$$

Now we know the velocity components in each of the three axes. Together they will combine to form the velocity vector of our star!

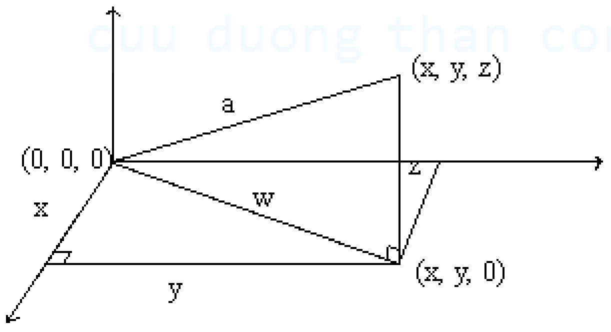
Once we know the velocity components, we can then proceed to calculate the (x,y,z) position of the star by using the following system of hopefully familiar equations:

$$d_x = v_x t \tag{14}$$

$$d_y = v_y t \tag{15}$$

$$d_z = v_z t \tag{16}$$

If you are having a difficult time visualizing the concept of vector components, look at figure **`3D Vector`**.



`3D Vector`

If you can visualize a star at point (x, y, z), then the line from the origin to (x, y, z) is the vector that represents the force due to gravity. In this figure, this is the line labelled "a". We can't simply use this direct line in our calculations, but must resolve the gravitation vector into its x, y, and z components as illustrated above. In physics, we would use trigonometry to do the component calculations. However, trigonometry is nothing more than the ratios of the lengths of the sides of a triangle. So, in our programs to simulate star dynamics we will use distances between the corresponding x, y, and z components as well as the Pythagorean distance between two stars to calculate the components of the gravitation vectors.

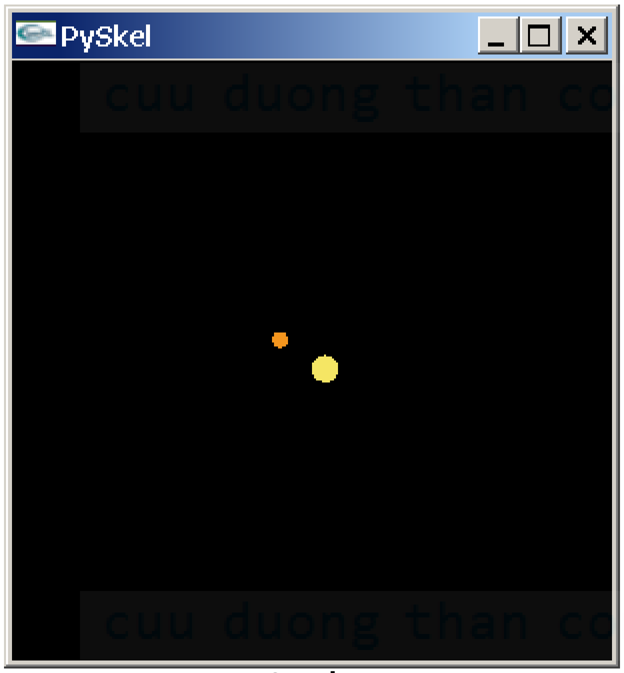
So, all those formulas you were forced to learn in math and science classes actually are useful for something! Let's see if we can put them together and write a program to simulate a simple 2 star system. Type in the following program and save it as **2body.py**:

```
# 2body.py
# a 2 star system based on Piet Hut
# and Jun Makino's MSA text with
# Modifications by Stan Blank for
# use in Python OpenGL/GLUT

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

import psyco
psyco.full()

#  Set the width and height of the window

global width
global height

#  Initial values

width = 500
height = 500

# initial values for position, velocity components, and time
# increment

global vx1, vy1, vz1, x1, y1, z1, r2, r3, ax1, ay1, az1, dt
global vx2, vy2, vz2, x2, y2, z2, ax2, ay2, az2, G

# initial x,y,z positions for both stars

x1 = 1.0
y1 = 0.0
z1 = 0.0
x2 = -1.0
y2 = 0.0
z2 = 0.0

# initial vx,vy,vz velocities for both stars

vx1 = 0.0
```

```
vy1 = -0.128571428
vz1 = 0.0
vx2 = 0.0
vy2 = 0.3
vz2 = 0.0

# initial masses for both stars

m1 = 0.7
m2 = 0.3
rad1 = 0.1*m1
rad2 = 0.1*m2

# arbitrary "Big G" gravitational constant

G = 1.0

# calculate distance and r**3 denominator for universal
# gravitation

r2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2)
r3 = r2*sqrt(r2)

# calculate acceleration components along x,y,z axes
# First for m1

ax1 = -G*(x1-x2)*m2/r3
ay1 = -G*(y1-y2)*m2/r3
az1 = -G*(z1-z2)*m2/r3

# now for m2

ax2 = -G*(x2-x1)*m1/r3
ay2 = -G*(y2-y1)*m1/r3
az2 = -G*(z2-z1)*m1/r3

# This value keeps a smooth orbit on my workstation
# Smaller values slow down orbit, higher values speed up orbit

dt = 0.001

def init():
     glClearColor(0.0, 0.0, 0.0, 1.0)

def reshape(  w,  h):

     # To insure we don't have a zero height

     if h==0:
          h = 1

     #  Fill the entire graphics window!
     glViewport(0, 0, w, h)

     #  Set the projection matrix... our "view"

     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     # Set how we view the world and position our eyeball

     gluPerspective(45.0, 1.0, 1.0, 1000.0)

     #  Set the matrix for the object we are drawing

     glMatrixMode(GL_MODELVIEW)
     glLoadIdentity()

     # Place the camera position, the direction of view
     # and which axis is UP

     gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)


def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'

     if key == chr(27):
          sys.exit()
     if key == "q":
          sys.exit()

def orbits():
     global vx1, vy1, vz1, x1, y1, z1, r2, r3, ax1, ay1, az1
     global vx2, vy2, vz2, x2, y2, z2, ax2, ay2, az2

     # calculate front half of velocity vector components

     vx1 += 0.5*ax1*dt
     vy1 += 0.5*ay1*dt
     vz1 += 0.5*az1*dt
     vx2 += 0.5*ax2*dt
     vy2 += 0.5*ay2*dt
     vz2 += 0.5*az2*dt

     # calculate x,y,z positions for both stars

     x1 += vx1*dt
     y1 += vy1*dt
     z1 += vz1*dt
     x2 += vx2*dt
     y2 += vy2*dt
     z2 += vz2*dt
```

```
      # calculate the new r**3 denominator for each star position

      r2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2)
      r3 = r2*sqrt(r2)

      # calculate the new acceleration components

      ax1 = -G*(x1-x2)*m2/r3
      ay1 = -G*(y1-y2)*m2/r3
      az1 = -G*(z1-z2)*m2/r3
      ax2 = -G*(x2-x1)*m1/r3
      ay2 = -G*(y2-y1)*m1/r3
      az2 = -G*(z2-z1)*m1/r3

      # calculate the back half velocity components

      vx1 += 0.5*ax1*dt
      vy1 += 0.5*ay1*dt
      vz1 += 0.5*az1*dt
      vx2 += 0.5*ax2*dt
      vy2 += 0.5*ay2*dt
      vz2 += 0.5*az2*dt

      #send calculated x,y,z star positions to the display

      glutPostRedisplay()

def plotfunc():
      glClear(GL_COLOR_BUFFER_BIT)

      # plot the first star (m1) position

      glPushMatrix()
      glTranslatef(x1,y1,z1)
      glColor3ub(245, 230, 100)
      glutSolidSphere(rad1, 10, 10)
      glPopMatrix()

      # plot the second star (m2) position

      glPushMatrix()
      glTranslatef(x2,y2,z2)
      glColor3ub(245, 150, 30)
      glutSolidSphere(rad2, 10, 10)
      glPopMatrix()

      # swap the drawing buffers

      glutSwapBuffers()

def main():
        global width
        global height

        glutInit(sys.argv)
        glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE)
        glutInitWindowPosition(100,100)
        glutInitWindowSize(width,height)
        glutCreateWindow("2 Body Problem")
        glutReshapeFunc(reshape)
        glutDisplayFunc(plotfunc)
        glutKeyboardFunc(keyboard)
        glutIdleFunc(orbits)

        init()
        glutMainLoop()

main()
```

If everything is typed correctly, when you run the program you should see two spherical objects in elliptical orbits passing near each other in the center of the graphics window. Figure **2Body** illustrates this scene:



2Body

If the simulation appears to be running too slowly, then reduce the **dt** variable by a power of 10, from **dt = 0.001** to **dt = 0.005** and run the program again.

Now for an explanation of the program (this should be interesting)! The first part of the code contains nothing new or unique with the exception of the larger number of global variables needed in the computational functions. Each of these global variables is given an initial value and these initial values contribute to the particular orbit displayed.

```
# initial x,y,z positions for both stars

x1 = 1.0
y1 = 0.0
z1 = 0.0
x2 = -1.0
y2 = 0.0
z2 = 0.0
```

We initially set the position components **`x1 = 1.0`** and **`x2 = -1.0`** and all other position components at 0.0. This will begin the simulation with both stars located on the x-axis at postive +1.0 and -1.0 units respectively. The initial velocity components are as follows:

```
# initial vx,vy,vz velocities for both stars

vx1 = 0.0
vy1 = -0.128571428
vz1 = 0.0
vx2 = 0.0
vy2 = 0.3
vz2 = 0.0
```

The first more massive star (m1) was given a downward y-axis component of **`vy1 = -0.128571428`** and the second less massive star (m2) was given and upward component y-axis component of **`vy2 = 0.3`**. All other velocity components were set at 0.0. These are not arbitray values for the velocity components, but were calculated beforehand to produce a nice stable orbit.

In the next section, we assign initial masses to the two stars m1 and m2 respectively. We also calculate a radius for both stars based on the individual star masses. When we display the stars in our graphics window, the radius of the spheres will provide some indication of their relative masses. In other words, a larger star will have a larger mass.

```
# initial masses for both stars

m1 = 0.7
m2 = 0.3
rad1 = 0.1*m1
rad2 = 0.1*m2
```

We also will assume a gravitational constant of **`G = 1.0`** (this is OUR simulation, we can do what we want!). The result will be a simulation of gravitational effects that differ only in scale from the "real" universe.

```
# arbitrary "Big G" gravitational constant

G = 1.0
```

Now we make some initial calculations to get the simulation headed down the correct path!

```
# calculate distance and r**3 denominator for universal
# gravitation

r2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2)
r3 = r2*sqrt(r2)

# calculate acceleration components along x,y,z axes
# First for m1

ax1 = -G*(x1-x2)*m2/r3
ay1 = -G*(y1-y2)*m2/r3
az1 = -G*(z1-z2)*m2/r3

# now for m2

ax2 = -G*(x2-x1)*m1/r3
ay2 = -G*(y2-y1)*m1/r3
az2 = -G*(z2-z1)*m1/r3
```

In the code above we performed some starting or "seed" calculations based on our initial values. The **r2** and **r3** variables calculate the r^3 denominator of Newton's universal gravitation equation in equations 6 through 10.[2] After finding the **r3** term for the denominator, we then calculate the initial **ax**, **ay**, and **az** acceleration components for each of the two bodies. The negative signs in front of "Big G" indicate that these accelerations are attractive rather than repulsive (in the force sense, not in appearance!). The subtraction of variables within parentheses after the "**G**" term, i.e. **(x1 – x2)** in the first calculation, finds the distance and the direction (+/-) between the bodies along whatever axis we choose. In this case, the **(x1 – x2)** term finds the distance between the two bodies along the x axis. The sign of this subtraction or distance calculation determines the direction of force. In effect, this distance is the r_x numerator term from equation 8. The same line of reasoning applies to the acceleration components along the y and z axes as well.

Now that we have the initial acceleration components, we can at least think about plotting orbits. First, we need to decide how to divide our orbit calculations into pieces or time slices. We'll do this as we did in the cannonball simulation (**pycannon.py**) and choose a small increment of time. In this program, we'll use:

```
dt = 0.001
```

We can alter this **dt** time increment value to speed up (larger time increment) or slow down (smaller time increment) the simulation. Smaller time increments result in more accurate calculations, but at the expense of a slower simulation rate. You will need to experiment on your own system to determine the best value for **dt**. Remember that we are actually working with calculus and difference equations in problems of this nature!

---

[2] Don't forget that we are multiplying Newton's Law of Gravitation by the unit vector to give a direction. This is why we must use the r^3 term instead of an r^2 term.

The **`def init()`** function is certainly familiar by now and in this program we are going to set a black background to match the background of space.

In the **`def reshape()`** function, we see something new:

```
gluPerspective(45.0, 1.0, 1.0, 1000.0)
```

and later:

```
gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)
```

Here are the beginnings of the journey into the land of 3D! Let's dissect the first statement, **`gluPerspective`**, and look at its arguments. The first value, in this case **`45.0`**, specifies the viewing angle of the observer. In other words, in this instance the field of view is 45.0 degrees. You can experiment with this value by making the number larger or smaller and see how the graphics display is affected. The second value in **`gluPerspective`**, in this case **`1.0`**, is the aspect ratio of the graphics presentation. A value of 1.0 means that the ratio of the width to height in the viewing window equals 1. If we change this value to something larger than 1.0, the display is "squeezed" from both sides. If we use a value smaller than 1.0, the display is "squashed" from above and below. It is possible to use the aspect ratio argument to adjust the display for a window resize/reshape, but we will not do that here.

The final two arguments specify the distance from our eye to the near and far clipping planes. A clipping plane marks the boundary of the graphics window. Should an object be drawn outside these boundaries, we will not be able to see what we have drawn. Both values must be positive and the near plane must be closer to us than the far plane. Again, you can experiment with these values to see how the display is changed. For example, using 0.0 for the near clipping plane results in the stars being sliced so that all we see is a cross-section. The reason for this behavior would be due to the fact that our eye would be in the near clipping plane rather than in front of it. This would be a bit like sitting IN the plane of the screen at a movie theater. It is much better to sit several rows away from the screen!

The **`gluLookAt`** statement appears to be more complicated, but it actually is rather easy to understand. The first 3 arguments, in this case **`0.0, 0.0, 5.0`**, specify the x, y, and z coordinates of your eye. We are familiar with 2 dimensional x, y coordinates from algebra and from this text. Adding a third dimension is as simple as adding another axis that comes out of the screen toward you and into the screen away from you. This axis is called the z axis and we can locate any point in space by giving the x, y, and z coordinates of that point. With the eye coordinates of (0.0, 0.0, 5.0), our eye is 5.0 units away from (or out of) the screen ON the z axis. The second set of 3 arguments, in this case **`0.0, 0.0, 0.0`**, locate the point in space that your eye is looking toward. The point (0.0, 0.0, 0.0) is the origin, so you are looking at the origin. That's simple! The final set of 3 arguments, **`0.0, 1.0, 0.0`**, simply tell the display which way is up. In this case, the y coordinate equals 1.0, so the positive y axis is considered to be up.

`gluLookAt` is a very powerful function! We can use it to actually move or fly through a graphics scene or display by changing the eye location, the point we are viewing, and the "up" direction. We will make use of this feature a bit later in the text, but keep this functionality in mind when you create your own programming projects or games.

The keyboard function is simple in that it allows us to exit the running simulation by either pressing the "Esc" key or the "q" key. Later we will use keyboard functions coupled with the powerful **`gluLookAt`** statement to fly through a more complex star cluster simulation.

The real "dirty work" of this program is found in the **`def orbits()`** function. The first calculations we must make are some velocity component calculations based on equations 11 – 13. If you are wondering why we are only calculating something called the "front half" of the velocity vector, the reason involves mathematics and calculation errors. When doing numerical calculus with difference equations, errors creep into the calculations rather quickly. We can get away with the small errors that enter into a short-lived simulation such as we created in the **pycannon.py** program. However, in simulations that are designed to run for extended periods of time, the errors can grow unacceptably large very quickly. To avoid this problem, we split the velocity calculation into 2 parts. The first part is calculated from the previous acceleration components and the second part is calculated from the new accleration components. Doing the velocity calculations in this manner provides a form of averaging and the errors are minimized. Now back to the program...

```
# calculate front half of velocity vector components

vx1 += 0.5*ax1*dt
vy1 += 0.5*ay1*dt
vz1 += 0.5*az1*dt
vx2 += 0.5*ax2*dt
vy2 += 0.5*ay2*dt
vz2 += 0.5*az2*dt
```

Once we know the velocity components, we can then find the (x, y, z) components of the star using equations 14 – 16. We are using the **+=** operator both in the velocity calculations above and in the position calculations below because we need to base new positions and velocities on the previous values for these quantities.

```
# calculate x,y,z positions for both stars
```

```
x1 += vx1*dt
y1 += vy1*dt
z1 += vz1*dt
x2 += vx2*dt
y2 += vy2*dt
z2 += vz2*dt
```

Since the starts are moving with respect to each other, the distances between the stars change and we must calculate a new value for the r3 denominator at each time step.

```
# calculate the new r**3 denominator for each star position

r2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2)
r3 = r2*sqrt(r2)
```

Unlike the positions and velocities, we calculate new values for the gravitational acceleration components "from scratch" during each time step based on the masses, distances, and directions of the stars with respect to other.

```
# calculate the new acceleration components

ax1 = -G*(x1-x2)*m2/r3
ay1 = -G*(y1-y2)*m2/r3
az1 = -G*(z1-z2)*m2/r3
ax2 = -G*(x2-x1)*m1/r3
ay2 = -G*(y2-y1)*m1/r3
az2 = -G*(z2-z1)*m1/r3
```

After the new acceleration components have been determined, we use them to calculate the "back half" of the velocity components. These velocity components and acceleration components will be used again as initial values when we enter the **def orbits()** function during the next time slice.

```
# calculate the back half velocity components

vx1 += 0.5*ax1*dt
vy1 += 0.5*ay1*dt
vz1 += 0.5*az1*dt
vx2 += 0.5*ax2*dt
vy2 += 0.5*ay2*dt
vz2 += 0.5*az2*dt
```

Finally, we want to update the display, so we issue a command to send the new position coordinates to the display function.

```
#send calculated x,y,z star positions to the display

glutPostRedisplay()
```

The display function is similar to bouncing ball program we wrote in the last section:

```
def plotfunc():
      glClear(GL_COLOR_BUFFER_BIT)

      # plot the first star (m1) position
      glPushMatrix()
      glTranslatef(x1,y1,z1)
      glColor3ub(245, 230, 100)
      glutSolidSphere(rad1, 10, 10)
```

```
        glPopMatrix()

        # plot the second star (m2) position
        glPushMatrix()
        glTranslatef(x2,y2,z2)
        glColor3ub(245, 150, 30)
        glutSolidSphere(rad2, 10, 10)
        glPopMatrix()

        # swap the drawing buffers
        glutSwapBuffers()
```

Once again we are using the **`glPushMatrix()`** and **`glPopMatrix()`** functions to keep the motions of the two stars independent of each other with the exception of their mutual gravitational effects. The colors of the stars are arbitrary and the radius of each star was calculated in the initial portion of the program and is based on the masses of the stars. After the stars are displayed at the proper coordinates, we issue the **`glutSwapBuffers()`** statement for smooth animation.

The def main(): function is familiar. Note again the **`GLUT_DOUBLE`** argument within the **`glutInitDisplayMode`** function to allow for double buffered animation. In the next section we will move from a 2 Body simulation to a 3 Body simulation, but first let's explore the current program by doing some Exercises. Oh, I forgot to mention that even though this chapter is entitled "2D Animation", with the addition of a z axis, we are really doing 3D simulations at this point. However, unless we specifically specify a z component for position or velocity, there will not be a z component for acceleration and we will have a 2D orbit system.

Exercises

1) The initial conditions in the simulation are such that the stars orbit each other in a pleasing manner with the motions of both describing well-formed ellipses. Try varying the initial positions of the stars in the **`x1, y1, z1`** and **`x2, y2, z2`** assignment statements at the beginning of the program.

2) The initial velocities are also open for experimentation. Change the initial velocities assigned to the **`vx1, vy1, vz1`** and **`vx2, vy2, vz2`** assignment statements to other values. You might try implementing a random number for each variable. How would you do this?

3) Experiment with the gravitational constant **`G`**. Try changing its value to something larger and/or smaller than 1.0 and see how the behavior of the star system changes. Make certain that you keep the same initial position and velocity parameters for each new value of **`G`** so that changes in the orbits will be more easily noticed. If you change BOTH the initial positions and velocities as well as the value of **`G`**, then you will not know which of the values caused the behaviors you see on the screen.

4) Comment the **`glClear`** statement in the **`def plotFunc`**(): function as follows:

```
#glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
```

What do you think will happen when you run the program now? Make certain you rememember to remove the comment!

5) Experiment with the **`gluLookAt`** function and see if you can "Look At" the orbits from a different viewpoint. Try to figure out how to move closer and then farther away.

6) Look up "NBody problem" online. Also, go to Piet Hut and Jun Makino's "Art of Computational Science" website (**www.artcompsci.org**) . I am personally involved in the "Moving Stars Around" (MSA) project, so make certain to visit the MSA area of the website!

7) There is nothing to prevent you from altering the computational part of the program by changing the formulas or altering how the gravitational forces are calculated. However, doing so will violate the known laws of physics and will invalidate the simulation. There is nothing inherently wrong with this, but understand that simulations are designed to model the interactions between objects in the universe. If we knowingly change the known laws of physics, then we will no longer be modeling OUR universe. As long as you understand this concept, then feel free to change the program however you like. I strongly recomment that you keep the original copy of the **`2Body.py`** program intact and simply rename the alternate universe copy.

Section 8.4 The REAL 3 Body Problem

In the previous section we were able to simulate the orbits of 2 stars interacting with each other through their gravitational forces. It should be relatively simple to add a 3rd star to the mix and see what happens... or should it be so easy? In a 2 body simulation, each star is affected only be its companion. In a 3 body simulation, each star is influenced by the sum of the interactions with BOTH of its companions. As we shall see, this is not so simple and requires more code than we might think. Remember that the 3 body problem is not solvable by a closed form equation. The best we can do is simulate the orbits by calculating the acceleration, velocity, and position components. Prior to the computer, this was a very difficult and labor-intensive task. Such calculations and resulting simulation is much simpler now that we have such enormous computing power at our fingertips.

The following listing is a 3 body simulation. You can, to some extent, use the **`2body.py`** program you've already written. Make certain to immediately change the name (Save As) of the program to **`3body.py`** or something similar. I will make no claim for the elegance or efficiency of this program. It works properly and getting a program to work properly is the foremost consideration of any programmer. Once a program is functioning correctly, then we can worry about elegance and efficiency if we so choose! More on efficiency later. For now, here is a 3 body simulation:

```
# 3body.py
# a 3 star system based on Piet Hut
# and Jun Makino's MSA 2 body text
# with 3 body modifications by Stan Blank
# for use in Python OpenGL/GLUT

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
import sys

import psyco
psyco.full()

#  Set the width and height of the window

global width
global height

#  Initial values for window width and height

width = 500
height = 500

# global variables for position, velocity  and
# acceleration components, time increment, and Gravity
```

```
global vx1, vy1, vz1, x1, y1, z1, ax1, ay1, az1
global vx2, vy2, vz2, x2, y2, z2, ax2, ay2, az2
global vx3, vy3, vz3, x3, y3, z3, ax3, ay3, az3, dt, G

# Initial values for position components in x,y,z space
# for each of the 3 star masses.  Note that the z-axis
# position is zero, so there is no z-component.  This is
# a 2D simulation at this point.

x1 = 1.0
y1 = 1.0
z1 = 0.0
x2 = -1.0
y2 = -1.0
z2 = 0.0
x3 = 0.50
y3 = -1.0
z3 = 0.0

# Initial values for velocity components in x,y,z space

vx1 = 0.0
vy1 = 0.0
vz1 = 0.0
vx2 = 0.0
vy2 = 0.0
vz2 = 0.0
vx3 = 0.0
vy3 = 0.0
vz3 = 0.0

# Initial acceleration components

ax1 = 0.0
ay1 = 0.0
az1 = 0.0
ax2 = 0.0
ay2 = 0.0
az2 = 0.0
ax3 = 0.0
ay3 = 0.0
az3 = 0.0

# Initial star masses

m1 = 0.7
m2 = 0.4
m3 = 0.5

# Gravitational Constant

G = 1.0
```

```
# radius of stars used in the plotFunc function

rad1 = 0.2*m1
rad2 = 0.2*m2
rad3 = 0.2*m3

# Calculate r**3 denominators for 3 Body Gravitation
# More complex because the motion of EACH star depends
# on where the other two stars are located!

r12 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2)
r312 = r12*sqrt(r12)
r13 = (x1-x3)*(x1-x3) + (y1-y3)*(y1-y3) + (z1-z3)*(z1-z3)
r313 = r13*sqrt(r13)
r23 = (x2-x3)*(x2-x3) + (y2-y3)*(y2-y3) + (z2-z3)*(z2-z3)
r323 = r23*sqrt(r23)

# Calculate the initial accelerations
# MUCH more complex than 2 Body dynamics
# Because each star must use the combined forces
# due to gravity of the other 2 stars.
# This is why there are TWO ax1, etc statements.

ax1 += -G*(x1-x2)*m2/r312
ax1 += -G*(x1-x3)*m3/r313
ay1 += -G*(y1-y2)*m2/r312
ay1 += -G*(y1-y3)*m3/r313
az1 += -G*(z1-z2)*m2/r312
az1 += -G*(z1-z3)*m3/r313
ax2 += -G*(x2-x1)*m1/r312
ax2 += -G*(x2-x3)*m3/r323
ay2 += -G*(y2-y1)*m1/r312
ay2 += -G*(y2-y3)*m3/r323
az2 += -G*(z2-z1)*m1/r312
az2 += -G*(z2-z3)*m3/r323
ax3 += -G*(x3-x2)*m2/r323
ax3 += -G*(x3-x1)*m1/r313
ay3 += -G*(y3-y2)*m2/r323
ay3 += -G*(y3-y1)*m1/r313
az3 += -G*(z3-z2)*m2/r323
az3 += -G*(z3-z1)*m1/r313

# This value keeps a smooth orbit on my workstation
# Smaller values slow down the orbit, higher values speed things
# up

dt = 0.001

def init():
     glClearColor(0.0, 0.0, 0.0, 1.0)
     glEnable(GL_DEPTH_TEST)
```

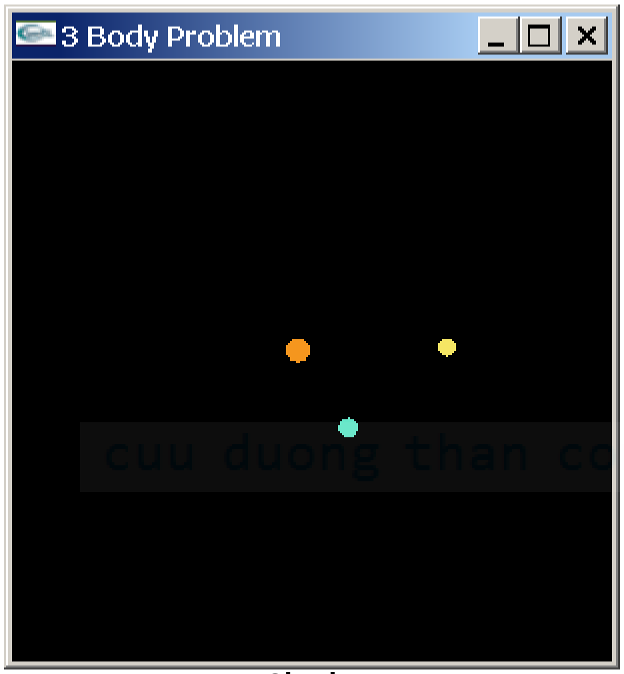
```
def reshape(  w,  h):

     # To insure we don't have a zero height

     if h==0:
          h = 1

     #  Fill the entire graphics window!

     glViewport(0, 0, w, h)

     #  Set the projection matrix... our "view"

     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     gluPerspective(45.0, 1.0, 1.0, 1000.0)

     glMatrixMode(GL_MODELVIEW)
     glLoadIdentity()

     gluLookAt(0.0, 0.0, 8.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)

def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'

     if key == chr(27):
          sys.exit()
     if key == "q":
          sys.exit()

def orbits():
     global vx1, vy1, vz1, x1, y1, z1, r2, r3, ax1, ay1, az1
     global vx2, vy2, vz2, x2, y2, z2, ax2, ay2, az2
     global vx3, vy3, vz3, x3, y3, z3, ax3, ay3, az3

     # More complex due to 3 Body instead of simply 2 Body
     # interactions.  This is the first half of the velocity
     # Calculations.  Known as Leap Frog!

     vx1 += 0.5*ax1*dt
     vy1 += 0.5*ay1*dt
     vz1 += 0.5*az1*dt
     vx2 += 0.5*ax2*dt
     vy2 += 0.5*ay2*dt
     vz2 += 0.5*az2*dt
     vx3 += 0.5*ax3*dt
     vy3 += 0.5*ay3*dt
     vz3 += 0.5*az3*dt

     # Calculate new positions
```

```
x1 += vx1*dt
y1 += vy1*dt
z1 += vz1*dt
x2 += vx2*dt
y2 += vy2*dt
z2 += vz2*dt
x3 += vx3*dt
y3 += vy3*dt
z3 += vz3*dt

# Reset acceleration components to zero.
# This is important!

ax1 = 0.0
ay1 = 0.0
az1 = 0.0
ax2 = 0.0
ay2 = 0.0
az2 = 0.0
ax3 = 0.0
ay3 = 0.0
az3 = 0.0

# Recalculate r**3 denominators

r12 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2)
r312 = r12*sqrt(r12)
r13 = (x1-x3)*(x1-x3) + (y1-y3)*(y1-y3) + (z1-z3)*(z1-z3)
r313 = r13*sqrt(r13)
r23 = (x2-x3)*(x2-x3) + (y2-y3)*(y2-y3) + (z2-z3)*(z2-z3)
r323 = r23*sqrt(r23)

# Calculate acceleration components from each body.
# We add or accumulate the acceleration components provided
# by each of the other two stars to arrive at ONE resultant
# Acceleration.  We avoid self-gravity!

ax1 += -G*(x1-x2)*m2/r312
ax1 += -G*(x1-x3)*m3/r313
ay1 += -G*(y1-y2)*m2/r312
ay1 += -G*(y1-y3)*m3/r313
az1 += -G*(z1-z2)*m2/r312
az1 += -G*(z1-z3)*m3/r313
ax2 += -G*(x2-x1)*m1/r312
ax2 += -G*(x2-x3)*m3/r323
ay2 += -G*(y2-y1)*m1/r312
ay2 += -G*(y2-y3)*m3/r323
az2 += -G*(z2-z1)*m1/r312
az2 += -G*(z2-z3)*m3/r323
ax3 += -G*(x3-x2)*m2/r323
ax3 += -G*(x3-x1)*m1/r313
```

```
     ay3 += -G*(y3-y2)*m2/r323
     ay3 += -G*(y3-y1)*m1/r313
     az3 += -G*(z3-z2)*m2/r323
     az3 += -G*(z3-z1)*m1/r313

     # Calculate the second half of the velocity components

     vx1 += 0.5*ax1*dt
     vy1 += 0.5*ay1*dt
     vz1 += 0.5*az1*dt
     vx2 += 0.5*ax2*dt
     vy2 += 0.5*ay2*dt
     vz2 += 0.5*az2*dt
     vx3 += 0.5*ax3*dt
     vy3 += 0.5*ay3*dt
     vz3 += 0.5*az3*dt

     #send x,y,z to the display

     glutPostRedisplay()

def plotfunc():
     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

     # Plot the position of m1

     glPushMatrix()
     glTranslatef(x1,y1,z1)
     glColor3ub(245, 150, 30)
     glutSolidSphere(rad1, 10, 10)
     glPopMatrix()

     # Plot the position of m2

     glPushMatrix()
     glTranslatef(x2,y2,z2)
     glColor3ub(245, 230, 100)
     glutSolidSphere(rad2, 10, 10)
     glPopMatrix()

     # Plot the position of m3

     glPushMatrix()
     glTranslatef(x3,y3,z3)
     glColor3ub(100, 230, 200)
     glutSolidSphere(rad3, 10, 10)
     glPopMatrix()
     glutSwapBuffers()

def main():
     global width
     global height
```

```
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE)
    glutInitWindowPosition(100,100)
    glutInitWindowSize(width,height)
    glutCreateWindow("3 Body Problem")
    glutReshapeFunc(reshape)
    glutDisplayFunc(plotfunc)
    glutKeyboardFunc(keyboard)
    glutIdleFunc(orbits)

    init()
    glutMainLoop()

main()
```

Once you have the program saved, run and debug the code if necessary.[3] If everything is running properly, you should see (at some point) a graphics display similar to the figure **3body** below. Watch as the stars dance with each other! Amazing, isn't it? I never tire of running this simulation and its variations. Even though the general format of this 3 body program is similar to the 2 body simulation, I think you will agree that this program is longer and more complicated than the relatively simple 2 body system. The physics you observe on the screen is MUCH more complex and the interactions between 3 bodies are far more complicated than the 2 body system. I think you can probably see why the 3 body problem is not generally solvable using mathematical equations except in a few specialized instances that we will view in the exercises.



3body

[3] The more complex the code, the more likely you will have errors! Get used to this… debugging is an essential skill for a successful programmer.

Looking at the listing of the 3 body program, you should immediately notice that there are many more variables and calculation statements than in the 2 body code. You would not think that simply adding one more star to the simulation would create this much extra work! It is even worse when we add a fourth, fifth, and sixth star as you will understand in a moment. Since I've already touched on the mathematics and physics of a gravitational simulation in previous sections, let's look at the code in the new 3 body program and see if we can understand why we need a more complex computational system.

Almost immediately we see MANY more global variables. This is too many to deal with efficiently and we'll fix that later, but for now we'll be programming using the brute strength and ignorance (BSI) approach. The BSI philosophy is easily stated: Get the program running correctly first and worry about elegance and efficiency later![4]

We need a unique variable for all position, velocity, and acceleration components for each star mass, hence the need for more variables. The initialization of the position and velocity variables is also understandable from the last program. Since there is an additional star mass to consider, it is reasonable to expect that we would require more initialization code. But then we encounter something new. Why must we initialize the acceleration components as in the following code?

```
# Initial acceleration components

ax1 = 0.0
ay1 = 0.0
az1 = 0.0
ax2 = 0.0
ay2 = 0.0
az2 = 0.0
ax3 = 0.0
ay3 = 0.0
az3 = 0.0
```

We did not have to do this in the 2 body program (although we could have!). The answer to the question is that we must accumulate all of the forces and combine them into a single acceleration component for each star. If we do not set an initial value for the individual acceleration components, we will generate an error during the first acceleration component calculations. We are setting the initial values of the acceleration components to zero.

After initializing the masses for each of the 3 stars and setting the value for **G**, we then calculate the radius of each star based on the star mass. We will use these radii in the **def plotfunc()** function to display relative star sizes.

```
rad1 = 0.2*m1
rad2 = 0.2*m2
rad3 = 0.2*m3
```

[4] I'm quite certain this approach is an anathema to the computer science wizards in the crowd. I am trying to care about that, but I am simply unable to bring myself to give a flying fig at this point.

Following the radius calculations, we must determine the initial r**3 denominators for each star pair interaction in the following code:

```
# Calculate r**3 denominators for 3 Body Gravitation
# More complex because the motion of EACH star depends
# on where the other two stars are located!

r12 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2)
r312 = r12*sqrt(r12)
r13 = (x1-x3)*(x1-x3) + (y1-y3)*(y1-y3) + (z1-z3)*(z1-z3)
r313 = r13*sqrt(r13)
r23 = (x2-x3)*(x2-x3) + (y2-y3)*(y2-y3) + (z2-z3)*(z2-z3)
r323 = r23*sqrt(r23)
```

Why is this code more complicated than the 2 body problem? In the 2 body simulation, we were concerned only with the single distance between the 2 stars. In the 3 body problem we must calculate forces using the distances between each of the star pairs. There are 3 stars, so there are 3 unique pairs of stars and therefore, 3 distances to calculate.[5] The variables are named to identify the star pair involved. For example, **`r12`** refers to the distance between stars **`m1`** and **`m2`**. Likewise, **`r312`** refers to the **`r**3`** denominator in Newton's law of gravitation for stars **`m1`** and **`m2`**. If we move to 4 bodies, the number of unique star pairs would be 6 (why?), necessitating doubling the number of equations above! How many equations would we need for 5 bodies if we keep on programming in this manner?

The next section of code is even more complex:

```
# Calculate the initial accelerations
# MUCH more complex than 2 Body dynamics
# Because each star must use the combined forces
# due to gravity of the other 2 stars.
# This is why there are TWO ax1, etc statements.

ax1 += -G*(x1-x2)*m2/r312
ax1 += -G*(x1-x3)*m3/r313
ay1 += -G*(y1-y2)*m2/r312
ay1 += -G*(y1-y3)*m3/r313
az1 += -G*(z1-z2)*m2/r312
az1 += -G*(z1-z3)*m3/r313
ax2 += -G*(x2-x1)*m1/r312
ax2 += -G*(x2-x3)*m3/r323
ay2 += -G*(y2-y1)*m1/r312
ay2 += -G*(y2-y3)*m3/r323
az2 += -G*(z2-z1)*m1/r312
az2 += -G*(z2-z3)*m3/r323
ax3 += -G*(x3-x2)*m2/r323
ax3 += -G*(x3-x1)*m1/r313
ay3 += -G*(y3-y2)*m2/r323
```

[5] The pairs are: m1-m2, m1-m3, and m2-m3

```
ay3 += -G*(y3-y1)*m1/r313
az3 += -G*(z3-z2)*m2/r323
az3 += -G*(z3-z1)*m1/r313
```

Why does it seem like we are duplicating statements? Why are there so many lines of code? Let's look at the first 2 lines in the section. These lines calculate the x acceleration component of the **`m1`** star mass (i.e. this is why the variable is **`ax1`**):

```
ax1 += -G*(x1-x2)*m2/r312
ax1 += -G*(x1-x3)*m3/r313
```

Even though these lines look similar, you will see that they are not exactly alike. Why do we need two lines of code that appear to calculate the same thing? Remember that this is a 3 body problem and the movement of each star is affected by the sum of the gravitational forces of the other 2 stars. Therefore, to calculate the acceleration of an individual star, we must add the gravitational forces of BOTH of the other stars together to arrive at a vector[6] sum of the acceleration components for each star. Strictly speaking, the **`+=`** increment operator is only needed in the second statement in each pair of component calculations (Why?). We could simplify these two statements into one longer statement (or its algebraic equivalent) as follows:

```
ax1 = -G*(x1-x2)*m2/r312 - G*(x1-x3)*m3/r313
```

Do you see how this single statement performs the same task as the two statements above? We can save several lines of code this way, particularly in the **`def orbits()`** function as we will soon see.

After setting the time increment, **`dt`**, we then encounter the **`def init()`**, **`def reshape()`**, and **`def keyboard()`** functions. These functions are not much different than in the previous 2 body program. The **`def orbits()`** function is where the orbital acceleration, velocity, and position components are calculated. You will notice immediately that this function is much more lengthy and complex than in the 2 body problem. Why shouldn't it be? Adding an extra star mass increases the number of calculations needed to determine the motion of each individual star. However, we can shorten the **`def orbits()`** function by a few lines of code if we combine the acceleration component calculations as shown above. Note that we must set the acceleration components back to zero before we recalculate the new acceleration for each star. Positions and velocities are cummulative, but acceleration is not (Why?). If we have shortened the acceleration component calculations by combining lines of code (as illustrated above), we will not need to set the acceleration components to zero each time increment (Why?).

In case you are wondering why I used so many lines of acceleration component code in the original 3 body program rather than the somewhat more efficient method of combining code statements, remember that I am a teacher and sometimes teachers do strange things in order to teach concepts! In exercise 1, you will have the opportunity to shorten the **`3body.py`** code if you wish.

[6] Don't forget that a vector contains BOTH magnitude and direction information.

The **`def plotfunc()`** function includes plotting instructions for all 3 stars, which makes the code a few lines longer than in the 2 body system. However, the structure of the code remains the same.

In the **`3body.py`** example program contained in this text, the time increment is set to **`dt = 0.001`**. This time increment allows the simulation to run smoothly on my workstation, a 2.13 Ghz Centrino Dell M70 laptop purchased in 2005. If your computer is faster than mine, you may want to select a smaller time increment. If your computer is a bit older or slower (such as my slothful 900 Mhz desktop), then you may wish to change this **`dt`** variable to a larger time increment such as **`dt = 0.01`**. If you are able to use **`dt = 0.001`**, you will see a dance of 3 stars and eventually (be patient!), one of the stars (the yellow one) will slide away to the upper left leaving the remaining two stars orbiting each other and moving slowly together to the lower right. According to Piet Hut, this is a realistic situation. With 3 body dynamics, often one star will gain enough gravitational acceleration to leave the system. The fleeing star takes much of the kinetic energy of the triple star system with it. When such a scenario occurs, this may leave the remaining 2 stars orbiting each other in a binary gravitational embrace. If you use a value for **`dt`** other than **`0.001`**, the dynamics of the 3 body system will change slightly and you may not obtain the same result.

Exercises

1) OK, I said that it was possible to shorten the program by combining some of the statements into one line as long as it made sense algebraically. Now is your chance to do just that. Try to combine or eliminate lines of code and see if you can make the program shorter in length. Test the program after EACH change you make to verify that the program still runs correctly! One advantage to making a program shorter is that you MAY be able to lessen the number of calculations needed to solve the problem. If so, the program may execute more quickly.

2) The dynamics of the 3 body system are so sensitive to initial conditions that the slightest change in any of the starting parameters can result in enormous changes in the behavior of the stars. What you should now do is make changes in the initial values for the variables. You might first try varying the masses of the stars. Then you can try changing the gravitational constant **`G`**. Finally, you can alter the starting velocities and positions. I recommend that you make only one change at a time so that should an interesting behavior appear, you will know exactly what you did to get that behavior.

3) If you experimented enough in exercise 2, you may have noticed that from time to time, two stars approached either other and then zoomed away as if they were shot from a cannon. Why might this happen? You'll recall Newton's law of universal gravitation (equation 1):

$$F = G\frac{m_1 m_2}{r^2}$$

If the distance, ***r***, between two stars approaches zero, the force due to gravity will increase enormously, resulting in huge accelerations for the stars. This is unnatural because two stars that approached each other that closely in nature would collide. Our simulation does not take collisions into account. One way to help alleviate this problem is to adjust the time step when two stars approach too closely. A smaller time step results in more precise calculations and an apparent slowing of the system. More precise calculations will often help avoid some collision situations. Another possibility, but more difficult to implement, is to allow for the collision and combine the colliding masses into one larger object. See if you can figure out how to change the time step to a smaller value if the distance between two stars becomes too small. Remember to change the time increment back after the close approach. What are some drawbacks to changing the time increment?

4) Even though the 3 body problem is not solvable using mathematical methods, there are certain configurations that have been discovered experimentally[7] that are amazingly stable. One famous 3 body configuration involves the stars orbiting each other in a figure-8 pattern. To replicate this configuation, here is what you must do. First, set the initial positions of the stars as follows:

```
x1 = -0.97000436
y1 = 0.24308753
z1 = 0.0
x2 = 0.97000436
y2 = -0.24308753
z2 = 0.0
x3 = 0.0
y3 = 0.0
z3 = 0.0
```

Next, set the initial velocities of the stars:

```
vx1 = -0.93240737/2
vy1 = -0.86473146/2
vz1 = 0.0
vx2 = -0.93240737/2
vy2 = -0.86473146/2
vz2 = 0.0
vx3 = 0.93240737
vy3 = 0.86473146
vz3 = 0.0
```

Finally, set the masses of each star to **`1.0`** and set **`G = 1.0`**. If you have everything running correctly, you should see an amazing figure-8 orbital dance of stars!

5) Explore the stable 3 body orbit by changing the mass of one of the stars to 1.1 and see what happens. How long do the orbits remain stable? Experiment with other masses such as 1.01 and see how long the orbits remain stable. Try subtly different velocities and positions to see how such small changes affect the figure-8 orbit.

[7] Using computer simulations much as we are doing here. You can actually do REAL science on your computer. That is remarkable!

6) Look up "stable 3 body orbits" online and get a glimpse of what the professional astronomers are doing! There are some other interesting stable orbit configurations involving multiple star masses.

Section 8.5 From 3Body to NBody Using Arrays

Based on the complexity of the 3body.py program, you can probably imagine how difficult it would be to extend the simulation to 4, 5, or 6 bodies! How about 20 bodies? We would have to write a specific program for each number of stars that we wanted to simulate. This is clearly inefficient! Piet Hut has coauthored a textbook which describes the process of simulating a million stars in a single cluster.8 Can you imagine how such a program would be written? First, our simple desktop or laptop computers would not be able to run such a huge simulation due to the incredible number of calculations involved. However, if we were to make an attempt to write a million-body simulation based on what we have done so far, the amount of extra code needed would be several million more lines or thousands of new pages of code! This task is clearly impossible unless there are better methods of writing such complex simulations. There ARE better methods and that is the topic of this section. Rather than write several extra lines of code for each new star we add to the simulation, why not use an array of variables?[9] We can use arrays to store data about each star in our cluster simulation. This makes the code MUCH more efficient and allows us the flexibility to extend our star cluster simulation to as many stars as we wish. We probably can't simulate the dynamics of one million stars, but we can simulate 4, 5, 10 or 20 without much difficulty. By the way, you may be wondering why such a simulation is called an "n-body" simulation? Apparently that's what astrophysicists decided to call multi-body any code that represents a system of multiple bodies or stars . There is no special reason. "n" can represent any number, hence "nbody".

Here is the code for an nbody simulation. Save it as **nbody.py**.

```
# NBody Code
# for multiple stars
# based on Piet Hut and Jun Makino's
# MSA Text

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
from random import *
import sys

import psyco
psyco.full()

#  Set the width and height of the window

global width
global height
```

[8] Heggie, Douglas; Hut, Piet (2003) "The Gravitational MillionBody Problem: A Multidisciplinary Approach to Star Cluster Dynamics", Cambridge University Press
[9] If you don't remember what arrays are and how they work, see Section 6.3 "The Chaos Game".

```
#  Initial values for window width and height

width = 600
height = 600

# global variables for position, velocity  and
# acceleration components, time increment, and Gravity

global n, m, v, a, r, rad, G, dt

# Time increment

dt = 0.0001

# Gravitational Constant

G = 1.0

# Initial number of stars

n = 20

# Initialize arrays for mass, velocity, acceleration
# position, radius, and color

m = zeros(n+1, Float)
vx = zeros(n+1, Float)
vy = zeros(n+1, Float)
vz = zeros(n+1, Float)
ax = zeros(n+1, Float)
ay = zeros(n+1, Float)
az = zeros(n+1, Float)
rx = zeros(n+1, Float)
ry = zeros(n+1, Float)
rz = zeros(n+1, Float)
rad = zeros(n+1, Float)
colr = zeros(n+1, Float)
colg = zeros(n+1, Float)
colb = zeros(n+1, Float)

def init():
     global m, r, a, v, rad, colr, colg, colb
     glClearColor(0.0, 0.0, 0.0, 1.0)

     # Enable depth testing for true 3D effects

     glEnable(GL_DEPTH_TEST)

     # Add lighting and shading effects

     glShadeModel(GL_SMOOTH)
     lightdiffuse = [1.0, 1.0, 1.0, 1.0]
```

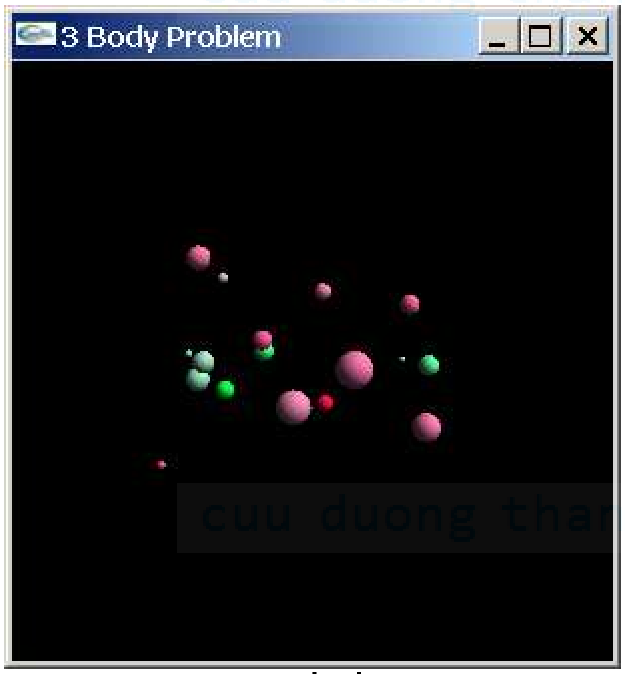
```
    lightposition = [1.0, 1.0, 1.0, 0.0]
    lightambient = [0.0, 0.0, 0.0, 1.0]
    lightspecular = [1.0, 1.0, 1.0, 1.0]

    # Turn on the light

    glLightfv(GL_LIGHT1, GL_DIFFUSE, lightdiffuse)
    glLightfv(GL_LIGHT1, GL_POSITION, lightposition)
    glLightfv(GL_LIGHT1, GL_AMBIENT, lightambient)
    glMaterialfv(GL_FRONT, GL_DIFFUSE, lightdiffuse)
    glMaterialfv(GL_FRONT, GL_SPECULAR, lightspecular)
    glEnable(GL_LIGHT1)
    glEnable(GL_LIGHTING)
    glEnable(GL_COLOR_MATERIAL)

    # Create a random set of n stars

    for i in range(1, n+1):

        m[i] = 500.*random() + 100.
        rad[i] = 0.0002*m[i]
        colr[i] = abs(sin(m[i]))
        colg[i] = abs(cos(m[i]))
        colb[i] = sqrt(abs(sin(m[i])*cos(m[i])))

    # Assign random positions to each star

    for i in range(1, n+1):
        rx[i] = cos(2*random()-1.25)*cos(5*random()-1.25)
        ry[i] = sin(2*random()-1.25)*cos(5*random()-1.25)
        rz[i] = sin(2*random()-1.25)

        # Set initial velocities and accelerations
        # of each star

        vx[i] = 0.0
        vy[i] = 0.0
        vz[i] = 0.0
        ax[i] = 0.0
        ay[i] = 0.0
        az[i] = 0.0

def orbits():
    global rx, ry, rz, vx, vy, vz, ax, ay, az

    # array calculations make things easier!

    for i in range(1,n+1):

        # First half of leapfrog algorithm

        vx[i] += 0.5*ax[i]*dt
        vy[i] += 0.5*ay[i]*dt
        vz[i] += 0.5*az[i]*dt

        rx[i] += vx[i]*dt
        ry[i] += vy[i]*dt
        rz[i] += vz[i]*dt

        ax[i] = 0.0
        ay[i] = 0.0
        az[i] = 0.0


        # Loop through ALL stars

        for j in range(1,n+1):

            # Do NOT act on self to avoid infinity!
            # Only calculate acceleration components
            # if we are working with OTHER stars

            if j != i:

                # Arrays are more efficient!
                # r2 calculation could be on 1 line
                # but it wouldn't fit the page margins

                r2 = (rx[i]-rx[j])*(rx[i]-rx[j])
                r2 += (ry[i]-ry[j])*(ry[i]-ry[j])
                r2 += (rz[i]-rz[j])*(rz[i]-rz[j])
                r3 = r2*sqrt(r2)

                ax[i] += -G*(rx[i]-rx[j])*m[j]/r3
                ay[i] += -G*(ry[i]-ry[j])*m[j]/r3
                az[i] += -G*(rz[i]-rz[j])*m[j]/r3

        # Second half of leapfrog algorithm

        vx[i] += 0.5*ax[i]*dt
        vy[i] += 0.5*ay[i]*dt
        vz[i] += 0.5*az[i]*dt

    glutPostRedisplay()

def reshape(  w,  h):

    # To insure we don't have a zero height

    if h==0:
        h = 1

    #  Fill the entire graphics window!
```

```
     glViewport(0, 0, w, h)

     #  Set the projection matrix... our "view"

     glMatrixMode(GL_PROJECTION)
     glLoadIdentity()

     gluPerspective(45.0, 1.0, 1.0, 1000.0)

     glMatrixMode(GL_MODELVIEW)
     glLoadIdentity()

     gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)

def keyboard(key, x, y):
     #  Allows us to quit by pressing 'Esc' or 'q'

     if key == chr(27):
          sys.exit()
     if key == "q":
          sys.exit()

def plotfunc():
     global m
     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

     # Again, hooray for arrays!

     for i in range(1,n+1):
          glPushMatrix()
          glTranslatef(rx[i],ry[i],rz[i])

          glColor3f(colr[i],colg[i],colb[i])
          glutSolidSphere(rad[i],20,20)
          glPopMatrix()

     glutSwapBuffers()

def main():
     global width
     global height

     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE)
     glutInitWindowPosition(100,100)
     glutInitWindowSize(width,height)
     glutCreateWindow("NBody Problem")
     glutReshapeFunc(reshape)
     glutDisplayFunc(plotfunc)
     glutKeyboardFunc(keyboard)
     glutIdleFunc(orbits)
```

```
    init()
    glutMainLoop()

main()
```

If your program is running correctly, you should see an initial graphics window that looks something like figure **nbody** on the next page. Your graphics window will probably not look exactly like the **nbody** figure because we chose random initial positions and sizes for the stars in our program. However, you should see 20 stars (unless one or more are hiding behind each other) and these stars will begin to move according to the gravitational attractions between them.

The wonderful thing about using arrays in this program is that we now have the flexibility to create a star cluster with as many stars as we wish (up to a practical limit, of course) without making major modifications to the program. If we want a system of 25 stars, we can simply set **`n = 25`** and then save and run the program. The result is a 25 star simulation! The drawback to using arrays is that arrays can be difficult to implement and understand. It is often hard to visual how arrays are utilized within a particular program and the **`nbody.py`** code is no exception. Different programming languages utilize arrays in different ways. Python is not particularly difficult in terms of array usage, but I find that array structures within C and BASIC are far easier for me to understand. This is my own personal opinion (and this is my textbook). Let's dig in and see what we can understand!



nbody

The first portion of the **`nbody.py`** program does not differ significantly from previous programs we've written. Very quickly, though, we see something new:

```
# Initialize arrays for mass, velocity, acceleration
# position, radius, and color

m = zeros(n+1, Float)
vx = zeros(n+1, Float)
vy = zeros(n+1, Float)
vz = zeros(n+1, Float)
ax = zeros(n+1, Float)
ay = zeros(n+1, Float)
az = zeros(n+1, Float)
rx = zeros(n+1, Float)
ry = zeros(n+1, Float)
rz = zeros(n+1, Float)
rad = zeros(n+1, Float)
colr = zeros(n+1, Float)
colg = zeros(n+1, Float)
colb = zeros(n+1, Float)
```

What is this? Each of the variables on the left side of the **=** sign has now been defined as an array. Remember that an array is an entire series or block of variables, all having the same variable name, but accessed by using a number or index. In the code above, not only are we defining several arrays, but we are also letting Python know how large the array needs to be, what kind of values will be stored in the array, and to start the array with nothing but zeroes. We are starting with zeros in order to have SOME value or number in each array element. Without a starting value in each array element, Python would most likely produce an error when we attempt to calculate and store a new value in the same location. We also specify that the arrays are floating point numbers. In order, the arrays store the following information:

m	– masses of the individual stars
vx	– the x velocity component of the individual stars
vy	– the y velocity component of the individual stars
vz	– the z velocity component of the individual stars
ax	– the x acceleration component of the individual stars
ay	– the y acceleration component of the individual stars
az	– the z acceleration component of the individual stars
rx	– the x coordinate of each star
ry	– the y coordinate of each star
rz	– the z coordinate of each star
rad	– the radius of each star
colr	– the red color parameter of each star
colg	– the green color parameter of each star
colb	– the blue color parameter of each star

Let's take a closer look at one of the code statements:

```
vx = zeros(n+1,Float)
```

In the statement above, the **vx** array will hold the x velocity component for each of the stars in our simulation. The **n+1** sets the size of the array. Why **n+1**? We simply can't have just **n**, the number of stars, as the number of elements in our array. Remember that we start counting array elements in Python (and most programming languages) at zero, so we must add 1 to the size of each array in order to conveniently start counting at 1.

Once the array is filled with values, we access each star's information as follows:

Star 1

x velocity component = vx[1]
y velocity component = vy[1]
z velocity component = vz[1]

Star 2

x velocity component = vx[2]
y velocity component = vy[2]
z velocity component = vz[2]

Star 3

x velocity component = vx[3]
y velocity component = vy[3]
z velocity component = vz[3]
.
.
.
Star 10

x velocity component = v[10]
y velocity component = v[10]
z velocity component = v[10]

Notice that each star's velocity components are stored in a **`vx[i]`**, **`vy[i]`**, or **`vz[i]`** array. The index of the array variable determines which star we are currently accessing. For example, the fourth star's z velocity component would be **`vz[4]`**. Why? What is the index for the y velocity component of the sixth star? A similar line of reasoning can be employed to understand the acceleration and position arrays.

Accessing the information contained in a simple array is relatively simple. One method would be to use a **`for`** loop ranging from 1 to n + 1. This is the method employed in the **`nbody.py`** code.

In the **`def init():`** function, we see some new statements.

```
# Enable depth testing for true 3D effects

glEnable(GL_DEPTH_TEST)

# Add lighting and shading effects

glShadeModel(GL_SMOOTH)
```

```
lightdiffuse = [1.0, 1.0, 1.0, 1.0]
lightposition = [1.0, 1.0, 1.0, 0.0]
lightambient = [0.30, 0.30, 0.30, 1.0]
lightspecular = [1.0, 1.0, 1.0, 1.0]

glLightfv(GL_LIGHT1, GL_DIFFUSE, lightdiffuse)
glLightfv(GL_LIGHT1, GL_POSITION, lightposition)
glLightfv(GL_LIGHT1, GL_AMBIENT, lightambient)
glMaterialfv(GL_FRONT, GL_DIFFUSE, lightdiffuse)
glMaterialfv(GL_FRONT, GL_SPECULAR, lightspecular)
glEnable(GL_LIGHT1)
glEnable(GL_LIGHTING)
glEnable(GL_COLOR_MATERIAL)
```

Objects intended to be displayed as solid or in 3D look much more realistic when lighting and depth effects are added. OpenGL has a built-in lighting system that allows a programmer to easily add lighting to a display scene. The first line, **`glEnable(GL_DEPTH_TEST)`** isn't exactly part of the lighting sequence, but allows for the proper display of 3D objects when one object, a star in this simulation, passes in front of another object. With **`GL_DEPTH_TEST`** enabled, objects that are supposed to be hidden by another object ARE hidden. This may not seem very important, but such an effect greatly adds to the illusion of 3D on a flat screen monitor. If you comment out this line and run the simulation again, you may be able to see the difference depth-testing makes in a 3D scene. Trust me when I tell you that this is a great feature in OpenGL! In the "old day", we had to actually calculate hidden lines and points in the code. That was not fun and tended to be difficult at best.

The next line, **`glShadeModel(GL_SMOOTH)`** allows the shading of an object to have a smooth appearance. For example, if two different colors are used at different points on a 3D object, **`GL_SMOOTH`** allows an interpolated, gradual blend of the two colors on all points in between.

The actual lighting occurs here:

```
lightdiffuse = [0.85, 0.85, 0.85, 0.0]
lightposition = [10.0, 10.0, 100.0, 0.0]
lightambient = [0.25, 0.25, 0.25, 0.0]
lightspecular = [1.0, 1.0, 1.0, 1.0]

glLightfv(GL_LIGHT1, GL_DIFFUSE, lightdiffuse)
glLightfv(GL_LIGHT1, GL_POSITION, lightposition)
glLightfv(GL_LIGHT1, GL_AMBIENT, lightambient)
glMaterialfv(GL_FRONT, GL_DIFFUSE, lightdiffuse)
glMaterialfv(GL_FRONT, GL_SPECULAR, lightspecular)
glEnable(GL_LIGHT1)
glEnable(GL_LIGHTING)
glEnable(GL_COLOR_MATERIAL)
```

The first three lines each represent an array of values that we will use to configure the behavior, placement, and background lighting. The first 3 values correspond to red, green, and blue color combinations. The fourth value is the alpha parameter, which deals with color transparency and blending. The **glLightfv** lines set the diffuse lighting, the x,y,z position of the light, and the ambient background light levels. The "**fv**" at the end of **glLightfv** means that we will be using a floating point array or vector[10] of values stored in a single array variable (such as **lightdiffuse**) to set the parameters of the **glLightfv** command. The **glMaterialfv** function allows the color of the 3D objects to be displayed properly when lighting is employed. We then turn on the lighting by specifying a light **glEnable(GL_LIGHT1)**, turning on the "power" to the light **glEnable(GL_LIGHTING)**, and enabling color **glEnable(GL_COLOR_MATERIAL)**. The **GL_SPECULAR** parameter triggers the "shininess" of an object. It is possible to create a mirror effect using OpenGL lighting. OpenGL has the capability for multiple lights (up to 8 at last check) and complex lighting effects. Objects can be illuminated from the exterior and from inside. Interior lighting (emission) can render a glowing appearance to objects. Lighting effects are both an art and a science. You should take the time to explore different lighting effects and make certain you research OpenGL lighting online! There are a LOT more options than we display in this simulation.

After we apply lighting, we then create our stars based on the number we assigned earlier, in this case, **n = 20**.

```
# Create a random set of n stars

for i in range(1, n+1):

     m[i] = 500.*random() + 100.
     rad[i] = 0.0002*m[i]
     colr[i] = abs(sin(m[i]))
     colg[i] = abs(cos(m[i]))
     colb[i] = sqrt(abs(sin(m[i])*cos(m[i])))
```

In this code block, we create n stars (**n = 20** in this example). Remember that when we use **for** loops, simply stating **range(1,n)** will not suffice since the loop will stop prior to reaching n. We must specify **range(1, n+1)** in order to create n stars. The mass of each star is randomly assigned a floating point value from 100 to 500. The radius of each star is calculated based on the mass and the red, green, and blue colors are assigned, again based on the mass of each star. The mathematical statements for color assignments are arbitrary.

```
# Assign random positions to each star

for i in range(1, n+1):
     rx[i] = cos(2*random()-1.25)*cos(5*random()-1.25)
     ry[i] = sin(2*random()-1.25)*cos(5*random()-1.25)
     rz[i] = sin(2*random()-1.25)

     # Set initial velocities and accelerations
     # of each star
```

[10] No, I'm not trying to confuse you. A vector in physics has magnitude and direction. Another definition for vector is a single row or column of values.

```
        vx[i] = 0.0
        vy[i] = 0.0
        vz[i] = 0.0
        ax[i] = 0.0
        ay[i] = 0.0
        az[i] = 0.0
```

Each star is assigned a random set of x, y, and z coordinates stored in the rx, ry, and rz arrays. The position assignment statements are the parametric equations for a sphere, so hopefully we'll get a somewhat spherical region of stars surrounding the origin. With small numbers for n, we obviously can't expect a perfect sphere. Once the stars are in place, the velocity and acceleration components are set to 0.0.

The true power of arrays is revealed in the **`def orbits():`** function. With arrays, we no longer have to worry about the complexity of our orbital calculations increasing every time we add an additional star mass. Here is the **`def orbits():`** function:

```
def orbits():
        global rx, ry, rz, vx, vy, vz, ax, ay, az

        # array calculations make things easier!

        for i in range(1,n+1):

                # first half of leapfrog algorithm

                vx[i] += 0.5*ax[i]*dt
                vy[i] += 0.5*ay[i]*dt
                vz[i] += 0.5*az[i]*dt

                rx[i] += vx[i]*dt
                ry[i] += vy[i]*dt
                rz[i] += vz[i]*dt

                ax[i] = 0.0
                ay[i] = 0.0
                az[i] = 0.0

                # Loop through ALL stars

                for j in range(1,n+1):

                        # Do NOT act on self to avoid infinity!
                        # Only calculate acceleration components
                        # if we are working with OTHER stars

                        if j != i:

                                # Arrays are more efficient!
```

```
                    # r2 calculation could be on 1 line
                    # but it wouldn't fit the page margins

                    r2 = (rx[i]-rx[j])*(rx[i]-rx[j])
                    r2 += (ry[i]-ry[j])*(ry[i]-ry[j])
                    r2 += (rz[i]-rz[j])*(rz[i]-rz[j])
                    r3 = r2*sqrt(r2)

                    ax[i] += -G*(rx[i]-rx[j])*m[j]/r3
                    ay[i] += -G*(ry[i]-ry[j])*m[j]/r3
                    az[i] += -G*(rz[i]-rz[j])*m[j]/r3

          # Second half of leapfrog algorithm

          vx[i] += 0.5*ax[i]*dt
          vy[i] += 0.5*ay[i]*dt
          vz[i] += 0.5*az[i]*dt

     glutPostRedisplay()
```

The first nine lines calculate the velocities and positions for the forward half of the leapfrog algorithm.[11] The **+=** operators accumulate the velocity and position components for each star in the simulation. The velocities and positions are running totals and depend on the previous values for each of the variables used. The acceleration components are NOT cummulative and must be recalculated each time the **def orbits()** function is called. This is why we set the acceleration components to 0.0 in this part of the loop. The next section is crucial:

```
     # Loop through ALL stars

     for j in range(1,n+1):

          # Do NOT act on self to avoid infinity!
          # Only calculate acceleration components
          # if we are working with OTHER stars

          if j != i:

               # Arrays are more efficient!
               # r2 calculation could be on 1 line
               # but it wouldn't fit the page margins

               r2 = (rx[i]-rx[j])*(rx[i]-rx[j])
               r2 += (ry[i]-ry[j])*(ry[i]-ry[j])
               r2 += (rz[i]-rz[j])*(rz[i]-rz[j])
               r3 = r2*sqrt(r2)

               ax[i] += -G*(rx[i]-rx[j])*m[j]/r3
```

[11] This is calculus, specifically we are numerically solving the difference equations for gravitational effects. Yes, leapfrog is its name.

```
                    ay[i] += -G*(ry[i]-ry[j])*m[j]/r3
                    az[i] += -G*(rz[i]-rz[j])*m[j]/r3
```

In order to calculate ALL the gravitational forces acting on each star, we must take each star individually (using the **i** loop) and move through all the OTHER stars in the cluster (using the **j** loop), adding all the acceleration components together to arrive a single value for the acceleration components acting on each individual star. We must make certain that we do NOT calculate a star's gravitational forces on itself to avoid a meaningless infinite result. That is the purpose of the **if j != i:** conditional statement. If **j** is NOT equal to **i**, then the star is "legitimate" and can be used for calculation.

Using arrays, we are able to perform all calculations more efficiently than in previous programs! The **for i in range(1, n+1):** loop chooses each star individually. The **for j in range(1,n+1):** starts a second or inner loop to look at all the OTHER stars in the cluster. The combination of **i** and **j** indexes in the calculations help perform the same computations that we did in the previous star cluster programs. The **r2** calculations are separated into several lines to avoid running over the page margins. We can do this by using the **+=** operator, which adds or accumulates the new calculated values for **r2** into the previous values. Notice the **(rx[i] – rx[j])** and similar statements in the **r2** and **ax**, **ay**, and **az** code lines. It is in these statements that we use loop indexes and arrays to calculate all the interactions between each star in our simulated cluster. Finally, we can compute the second half of the velocity components and call the **glutPostRedisplay()** command to update the graphics window.

The **def reshape(w, h):** and **def keyboard(key, x, y):** functions are similar to previous programs. This particular keyboard function allows us to exit the program using either the ESC or 'q' key. The **def plotfunc():** display function is much simpler than in the previous 3body program:

```
def plotfunc():
     global m
     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

     # Again, hooray for arrays!

     for i in range(1,n+1):
          glPushMatrix()
          glTranslatef(rx[i],ry[i],rz[i])

          glColor3f(colr[i],colg[i],colb[i])
          glutSolidSphere(rad[i],20,20)
          glPopMatrix()

     glutSwapBuffers()
```

Using arrays, we can simply loop through each of the stars (**n = 20** in this program) translating to the proper position and plotting the star using the previously calculated radius and color scheme. Nothing could be simpler!

I won't pretend to have present you with everything there is to know about using arrays. Arrays are a topic that require time and mental effort on your part to understand. The wonderful thing about arrays in this program is that we can select an arbitrary number of stars by changing ONE SIMPLE LINE OF CODE. If we want 50 stars, we set **`n = 50`** at the start of the program. If we want 100 stars, set **`n = 100`**. There is no theoretical limit to **`n`**, however there are practical limits determined by your computer and Python. It would be nice to simply specify **`n = 1000000`** to simulate one million stars. However, if your computer ran slowly with 100 stars, it might not run at all with 1000000 stars due to memory and processor limitations. Piet Hut and Jun Makino run very large star cluster simulations using a specialized computer call the GRAPE.[12] Look up Jun Makino and GRAPE using Google and see what information is available. The GRAPE is a true super-computer, but it is very specialized and works only with star cluster dynamics problems.

In the next section, we are going to add the ability to fly through this **`nbody.py`** program as if we were in a space ship. 3D navigation is an important function in terms of simulations and gaming.

Exercises

1) Experiment with lighting. How might you do this? You could change some of the values in this program or even better, look up OpenGL lighting online and figure out how lighting actually works. For example, see if you can make the stars actually GLOW like stars!

2) Increase the value for n until your computer begins to run slowly and/or badly. You will notice that instead of smooth animation, the updating becomes ragged and "jerky". You can also adjust the time increment (dt) by making it smaller to see if this helps the animation. What is the maximum star limit for relatively smooth animation on your computer?

3) Yes, you can try **`n = 1000000`**. What happens? Nothing? Do you think nothing is going on or are the number of calculations simply too huge to complete in a reasonable amount of time?

4) Feel free to experiment in other ways. For example, you can adjust the mass, radius, and color calculations if you choose.

5) How might you assign the initial values for position and velocity "on purpose" in order to replicate the stable figure 8 orbit from the last section? Hint: You can do something like this --> rx[3] = -0.234

6) Are there any stable orbit configurations of more than 3 bodies? Do some research online and see if you can find any “beasts”.

7) You do not have to build stars that are spherical and solid. Using wireframe stars should speed up the simulation somewhat. For example, using

[12] Jun Makino and Piet Hut developed the specialized GRAPE computer themselves!

`glutWireSphere(rad[i], 5, 5)` may make you program run a bit faster. You might even try cubes or tetrahedrons. What do orbiting teapots look like?

8) A great online resource for the nbody problem in addition to Piet Hut's website is Sverre Aarseth's great website **`www.sverre.com`** Sverre is one of the pioneers in computational nbody simulation.

Section 8.6 Navigating the Stars

One of the drawbacks of the **`nbody.py`** program is that we are limited in what we can view. Using the **`gluLookAt`** command we can set the position of our eye or camera at the start of the program, but what happens if the stars move out of our sight? It would be nice to have the capability of navigating or flying through the star cluster, changing our position and our view as needed. This would add great flexibility to our program and might even suggest the possibility of creating a (dare I say it?) game at some point in our programming journey! How can we add a navigation feature to our simulation? It really isn't too difficult. We can use keyboard routines to move our camera (OK… our spaceship) position via the **`gluLookAt`** function. **`gluLookAt`** is a very powerful command and is flexible enough to allow us to use it at any time and not just at program startup. I'm going to list the entire nbody.py program again, with the navigation functions added. You can simply reuse the program from the last section, rename it **`flynbody.py`** or something similar, and make the changes and additions where necessary. The result is interesting at the very least!

```
# Fly Through NBody Code
# for multiple stars
# based on Piet Hut and Jun Makino's
# MSA Text

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from Numeric import *
from random import *
import sys

import psyco
psyco.full()

#  Set the width and height of the window

global width
global height

#  Initial values for window width and height

width = 500
height = 500

# global variables for position, velocity  and
# acceleration components, time increment, and Gravity

global n, m, v, a, r, rad, G, dt

# gluLookAt variables

global x, y, z, lx, ly, lz
```

```
# Time increment

dt = 0.001

# Gravitational Constant

G = 1.0

# Initial number of stars

n = 20

# Initialize arrays for mass, velocity, acceleration
# position, radius, and color

m = zeros(n+1, Float)
vx = zeros(n+1, Float)
vy = zeros(n+1, Float)
vz = zeros(n+1, Float)
ax = zeros(n+1,Float)
ay = zeros(n+1,Float)
az = zeros(n+1, Float)
rx = zeros(n+1,Float)
ry = zeros(n+1,Float)
rz = zeros(n+1,Float)
rad = zeros(n+1,Float)
colr = zeros(n+1,Float)
colg = zeros(n+1,Float)
colb = zeros(n+1,Float)

def initgl():
     global m, r, a, v, rad, colr, colg, colb
     glClearColor(0.0, 0.0, 0.0, 1.0)

     # Enable depth testing for true 3D effects

     glEnable(GL_DEPTH_TEST)

     # Add lighting and shading effects

     glShadeModel(GL_SMOOTH)
     lightdiffuse = [1.0, 1.0, 1.0, 1.0]
     lightposition = [1.0, 1.0, 1.0, 0.0]
     lightambient = [0.0, 0.0, 0.0, 1.0]
     lightspecular = [1.0, 1.0, 1.0, 1.0]

     # Turn on the light

     glLightfv(GL_LIGHT1, GL_DIFFUSE, lightdiffuse)
     glLightfv(GL_LIGHT1, GL_POSITION, lightposition)
     glLightfv(GL_LIGHT1, GL_AMBIENT, lightambient)
     glMaterialfv(GL_FRONT, GL_DIFFUSE, lightdiffuse)
     glMaterialfv(GL_FRONT, GL_SPECULAR, lightspecular)
     glEnable(GL_LIGHT1)
     glEnable(GL_LIGHTING)
     glEnable(GL_COLOR_MATERIAL)

def init():
     # Create a random set of n stars

     for i in range(1, n+1):

          m[i] = 500.*random() + 100.
          rad[i] = 0.0002*m[i]
          colr[i] = abs(sin(m[i]))
          colg[i] = abs(cos(m[i]))
          colb[i] = sqrt(abs(sin(m[i])*cos(m[i])))

     # Assign random positions to each star

     for i in range(1, n+1):
          rx[i] = cos(2*random()-1.25)*cos(5*random()-1.25)
          ry[i] = sin(2*random()-1.25)*cos(5*random()-1.25)
          rz[i] = sin(2*random()-1.25)

          # Set initial velocities and accelerations
          # of each star

          vx[i] = 0.0
          vy[i] = 0.0
          vz[i] = 0.0
          ax[i] = 0.0
          ay[i] = 0.0
          az[i] = 0.0

def tilt(azim):
     # Tilts the camera/spaceship up and down

     global ly
     ly = sin(azim)
     lz = -cos(azim)
     glLoadIdentity()
     gluLookAt(x, y, z, x + lx, y + ly, z + lz, 0.0, 1.0, 0.0)

def latmove(mov):
     # Moves the camera/spaceship forward and backward
     # along the line of sight

     global x, y, z
     x += mov*lx*n/10
     y += mov*ly*n/10
     z += mov*lz*n/10
     glLoadIdentity()
```

```
    gluLookAt(x, y, z, x + lx, y + ly, z + lz, 0.0, 1.0, 0.0)

def swivel(theta):
    # Swivels the camera/spaceship left and right

    global lx, lz
    lx = sin(theta)
    lz = -cos(theta)
    glLoadIdentity()
    gluLookAt(x, y, z, x + lx, y + ly, z + lz, 0.0, 1.0, 0.0)

def orbits():
    global rx, ry, rz, vx, vy, vz, ax, ay, az

    # array calculations make things easier!

    for i in range(1,n+1):

        # First half of leapfrog algorithm

        vx[i] += 0.5*ax[i]*dt
        vy[i] += 0.5*ay[i]*dt
        vz[i] += 0.5*az[i]*dt

        rx[i] += vx[i]*dt
        ry[i] += vy[i]*dt
        rz[i] += vz[i]*dt

        ax[i] = 0.0
        ay[i] = 0.0
        az[i] = 0.0


        # Loop through ALL stars

        for j in range(1,n+1):

            # Do NOT act on self to avoid infinity!
            # Only calculate acceleration components
            # if we are working with OTHER stars

            if j != i:

                # Arrays are more efficient!
                # r2 calculation could be on 1 line
                # but it wouldn't fit the page margins

                r2 = (rx[i]-rx[j])*(rx[i]-rx[j])
                r2 += (ry[i]-ry[j])*(ry[i]-ry[j])
                r2 += (rz[i]-rz[j])*(rz[i]-rz[j])
                r3 = r2*sqrt(r2)
```

```
                    ax[i] += -G*(rx[i]-rx[j])*m[j]/r3
                    ay[i] += -G*(ry[i]-ry[j])*m[j]/r3
                    az[i] += -G*(rz[i]-rz[j])*m[j]/r3

           # Second half of leapfrog algorithm

           vx[i] += 0.5*ax[i]*dt
           vy[i] += 0.5*ay[i]*dt
           vz[i] += 0.5*az[i]*dt

      glutPostRedisplay()

def reshape(  w,  h):

      # To insure we don't have a zero height

      if h==0:
           h = 1

      #  Fill the entire graphics window!

      glViewport(0, 0, w, h)

      #  Set the projection matrix... our "view"

      glMatrixMode(GL_PROJECTION)
      glLoadIdentity()

      gluPerspective(45.0, 1.0, 1.0, 1000.0)

      glMatrixMode(GL_MODELVIEW)
      glLoadIdentity()

      gluLookAt(x, y, z, x + lx, y + ly, z + lz, 0.0, 1.0, 0.0)

def keyboard(key, x, y):
      #  Allows us to quit by pressing 'Esc' or 'q'

      if key == chr(27):
           sys.exit()
      if key == "q":
           sys.exit()

      # Reset view to original position
      if key == "z":
           zap()

# The specialkey function looks for arrow keys
def specialkey(key, x, y):
      global ang, updown, move

      # mode checks for SHIFT key
```

```
        mode = glutGetModifiers()

        # Left arrow key
        if key == GLUT_KEY_LEFT:
                ang = -0.01

        # Right arrow key
        if key == GLUT_KEY_RIGHT:
                ang = 0.01

        # Up arrow key
        if key == GLUT_KEY_UP:
                # If shift key is used, tilt upward
                if mode == GLUT_ACTIVE_SHIFT:
                        updown = -0.01
                # Otherwise move forward
                else:
                        move = 0.05

        # Down arrow key
        if key == GLUT_KEY_DOWN:
                # If shift key is used, tilt downward
                if mode == GLUT_ACTIVE_SHIFT:
                        updown = 0.01
                # Otherwise move backward
                else:
                        move = -0.05

def plotfunc():
        global m, move, angle, ang, azim, updown

        # moving around in 3D
        # Based on specialkeys

        # move forward and backward
        # along the line of sight
        if move != 0:
                latmove(move)
                move = 0

        # Swivel left and right
        if ang != 0.0:
                angle += ang*n/10
                swivel(angle)
                ang = 0.0

        # Tilt up and down
        if updown != 0.0:
                azim += updown
                tilt(azim)
                updown = 0.0
```

```
     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

     # Again, hooray for arrays!

     for i in range(1,n+1):
          glPushMatrix()
          glTranslatef(rx[i],ry[i],rz[i])

          glColor3f(colr[i],colg[i],colb[i])
          glutSolidSphere(rad[i],20,20)
          glPopMatrix()

     glutSwapBuffers()

def zap():
     global angle, ang, updown, azim, move
     global x, y, z, lx, ly, lz
     # Navigation variables

     angle = 0.0
     ang = 0.0
     updown = 0.0
     azim = 0.0
     move = 0.0

     # Initial values for eyeball position

     x = 0.0
     y = 0.0
     z = 10.0
     lx = 0.0
     ly = 0.0
     lz = -10.0

     # makes certain we set the view "straight ahead"
     swivel(0)

def main():
     global width
     global height

     glutInit(sys.argv)
     glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE)
     glutInitWindowPosition(100,100)
     glutInitWindowSize(width,height)
     glutCreateWindow("NBody Problem")
     glutReshapeFunc(reshape)
     glutDisplayFunc(plotfunc)
     glutKeyboardFunc(keyboard)
     glutSpecialFunc(specialkey)
     glutIdleFunc(orbits)
```

```
    initgl()
    init()
    zap()
    glutMainLoop()

main()
```

Assuming that everything has been correctly entered and properly saved, you should be able to run the program. When you do, you will probably not notice much difference between this program and the previous **nbody.py** simulation. However, if you press one of the arrow keys, you should immediately see a new feature. We can MOVE within the simulation environment! The instructions for movement are as follows:

The UP arrow key moves us forward along the line of sight
The DOWN arrow key moves us backward along the line of sight
The LEFT arrow key swivels the camera to the left
The RIGHT arrow key swivels the camera to the right
The SHIFT+UP arrow tilts the camera down as if we were diving
The SHIFT+DOWN arrow tilts the camera up as if we were climbing

Once a new orientation is reached, the UP and DOWN arrow keys move us along the new line of sight. The result is that we can travel anywhere within the simulation, even into the middle of the cluster! If you get lost, pressing the "z" key will "ZAP" you back to the original viewpoint. Neat, isn't it?

So how does this program differ from the previous nbody program? Let's look at the listing and see what we can find. Almost immediately we encounter some new global variables:

```
# gluLookAt variables

global x, y, z, lx, ly, lz
```

This set of variables suggests that we are going to do something new with the **gluLookAt** statement and we are, of course! Instead of a single **def init():** function as in previous programs, this new program divides the initialization up into a lighting function (**def initgl():**) and a function to create the stars, star masses, and initial positions (**def init():**). Strictly speaking, we do not have to do this, but sometimes separating discrete tasks into smaller functions makes a program listing more readable. In this case, we have two tasks: Lighting and star intialization. Why not create separate functions to perform each task? That is what we have done.

Following the initialization functions, we see three new functions:

```
def tilt(azim):
    # Tilts the camera/spaceship up and down
    global ly
    ly = sin(azim)
    lz = -cos(azim)
    glLoadIdentity()
    gluLookAt(x, y, z, x + lx, y + ly, z + lz, 0.0, 1.0, 0.0)
```

```
def latmove(mov):
     # Moves the camera/spaceship forward and backward
     # along the line of sight
     global x, y, z
     x += mov*lx*n/10
     y += mov*ly*n/10
     z += mov*lz*n/10
     glLoadIdentity()
     gluLookAt(x, y, z, x + lx, y + ly, z + lz, 0.0, 1.0, 0.0)

def swivel(theta):
     # Swivels the camera/spaceship left and right
     global lx, lz
     lx = sin(theta)
     lz = -cos(theta)
     glLoadIdentity()
     gluLookAt(x, y, z, x + lx, y + ly, z + lz, 0.0, 1.0, 0.0)
```

As you might be able to guess, the first function, **`def tilt(azim):`**, provides a method for tilting the camera up or down, much like tilting your head. The **`sin(azim)`** and **`–cos(azim)`** mathematical functions are used here because tilt would be measured in terms of an angle, represented by the variable **`azim`**[13] and tilting results in a shift of both the y and z positions of objects. The variable, **`azim`**, is "passed" to **`def tilt`** from the **`def specialkeys`** function elsewhere in the program. We can implement a tilt in our viewing by pressing both the SHIFT key AND the UP or DOWN arrow keys at the same time. Notice that once we have "tilted" and stored the numeric value for **`sin(azim)`** and **`–cos(azim)`** in the variables **`ly`** and **`lz`**, we call the **`gluLookAt`** function to load the new tilted view in our graphics window. For your information, the **`glLoadIdentity()`** statement just prior to the **`gluLookAt`** command uses the identity matrix[14] to set the current view so that any future changes will be based on our current position.

The second function, **`def latmove(mov):`**, provides lateral motion along the current viewing axis using the UP and DOWN arrow keys. In this function, the variable **`mov`** is passed from the **`def specialkeys`** function and results in a change in the x, y, and z coordinates of the camera. Once the new x, y, and z coordinates have been calculated, the **`gluLookAt`** function is called again to create a new scene and simulate motion forward and backward motion.

The **`def swivel(theta):`** function uses the LEFT and RIGHT arrow keys to pass an angle **`theta`** to the function. Using this angle, the camera can be swivelled from left to right in a similar fashion as in the **`def tilt`** function. This time, however, a swivel results in movements in the x and z coordinates of the objects in the display

[13] Representing azimuth, which is an angle between a reference point and a plane. An example would be the angle that the moon makes with the horizon as viewed by your eye.
[14] The identity matrix is a matrix with 1's along the upper left to lower right diagonal. Multiplying by the identity matrix produces the original matrix just like multiplying by one produces the original number.

window. Once the **`lx`** and **`lz`** variables are defined, they are sent to the gluLookAt command so the new view can be displayed.

The **`def orbits`** and **`def reshape`** functions are unchanged. The **`def keyboard`** function is a bit different, though. We have added a new keystroke command:

```
    # Reset view to original position
    if key == "z":
        zap()
```

If the "z" key is pressed, the **`zap()`** function is called. Let's look at the **`def zap():`** function now:

```
def zap():
    global angle, ang, updown, azim, move
    global x, y, z, lx, ly, lz
    # Navigation variables

    angle = 0.0
    ang = 0.0
    updown = 0.0
    azim = 0.0
    move = 0.0

    # Initial values for eyeball position

    x = 0.0
    y = 0.0
    z = 10.0
    lx = 0.0
    ly = 0.0
    lz = -10.0

    # makes certain we set the view "straight ahead"
    swivel(0)
```

The **`def zap`** function returns the simulation to the original viewpoint by resetting all **`gluLookAt`** variables back to their original values. Sending an angle of 0 to the **`swivel`** function turns the camera "straight ahead" to its original orientation.

A new set of commands is found in the **`def specialkeys`** function:

```
# The specialkey function looks for arrow keys

def specialkey(key, x, y):
    global ang, updown, move

    # mode checks for SHIFT key
    mode = glutGetModifiers()
```

```
     # Left arrow key
     if key == GLUT_KEY_LEFT:
          ang = -0.01

     # Right arrow key
     if key == GLUT_KEY_RIGHT:
          ang = 0.01

     # Up arrow key
     if key == GLUT_KEY_UP:
          # If shift key is used, tilt upward
          if mode == GLUT_ACTIVE_SHIFT:
               updown = -0.01
          # Otherwise move forward
          else:
               move = 0.05

     # Down arrow key
     if key == GLUT_KEY_DOWN:
          # If shift key is used, tilt downward
          if mode == GLUT_ACTIVE_SHIFT:
               updown = 0.01
          # Otherwise move backward
          else:
               move = -0.05
```

Built into GLUT is the capability of looking for the "special" keys on a keyboard, such as the ARROW keys, END, HOME, PAGE UP, PAGE DOWN, and SHIFT key combinations as well as the CTRL and ALT keys. I believe the **specialkeys** function is fairly self-explanatory. It is here that we are setting the values of variables that will be employed in creating movement. By changing the values assigned to these variables, we can alter how quickly we move. However, fast movement comes at the expense of smooth movements! Make note of the statement:

```
     mode = glutGetModifiers()
```

In this statement, we can check for keys that modify other keys. The SHIFT key is such a modifier. We can check for the presence of the SHIFT key by the value of the mode variable. If **mode == GLUT_ACTIVE_SHIFT**, then the shift key has been pressed.

The **def plotfunc():** function has some added lines of code at the beginning:

```
def plotfunc():
     global m, move, angle, ang, azim, updown

     # moving around in 3D
     # Based on specialkeys

     # move forward and backward
     # along the line of sight
```

```
    if move != 0:
        latmove(move)
        move = 0

    # Swivel left and right
    if ang != 0.0:
        angle += ang*n/10
        swivel(angle)
        ang = 0.0

    # Tilt up and down
    if updown != 0.0:
        azim += updown
        tilt(azim)
        updown = 0.0
```

It is here that we check to see if any of the movement keys have been pressed. If any of those keys have been active, then the variables **move**, **ang**, and **updown** will have been changed from 0.0 to a different value (equal to the amount of movement specified in the **def specialkeys** function). If any of the variables have a value other than zero, the appropriate movement function will be called and the new view position will be displayed. When the **latmove**, **swivel**, or **tilt** functions "do their thing", the program returns and sets the value of the variable (such as **move**) back to zero. If we fail to reset the variable to zero, continuous movement results. Try it! Place a comment in front of the **ang = 0.0** statement, save and run the program, and then press the left or right arrow key. What happens? There is nothing wrong with this behavior, but it might make you dizzy! The remainder of the **def plotfunc()** function is unchanged from the previous program.

The only other significant change is in the **def main():** function. If you will look closely, we've added the statement:

```
    glutSpecialFunc(specialkey)
```

This command will identify the function that captures the "special" keystrokes needed in this program. Also notice that prior to issuing the **glutMainLoop** statement, we have:

```
    initgl()
    init()
    zap()
```

in order to call the program startup functions for lighting, stars, and the proper view.

Exercises

1) Explore the movement routines in this program. See if you can change how the camera... or spaceship if you prefer, moves through space. Research the GLUT special keys online to see the possibilities for program control.

2) instead of the SHIFT + ARROW key combination for tilting, try to use the PageUp and PageDown keys instead.

3) What other movements can you think to employ? Can you control the position of the lighting using some keys, special or otherwise?

Chapter 9 3D and 3D Animation

The last chapter was basically an introduction to 2D animation, but I cheated a bit and added some 3D code to simulate star clusters. OpenGL was originally intended to create 3D objects, scenes, and animation, so after many pages of 2D topics, 3D is where we will concentrate our efforts until the end of the book. When discussing 3D objects and scenes, we need to realize that our monitor screen or laptop dispay is only a 2D surface. I know there are special glasses, headsets, screens, holographic, and CAVE/CUBE virtual reality display devices that create "true" 3D scenes, but most of these devices are not easily accessible for mere mortals. What we must be able to do in order to simulate 3D is to draw or render objects on the screen to provide the illusion of 3 dimensions. We can do this by depth testing so that near objects block or occlude farther objects. Perspective can be employed to make objects that are farther away appear smaller. Lighting and shading can create 3D effects and we can even use fog to make distant objects fade away. The most important OpenGL 3D effect is that we can have objects move or rotate in such a fashion that our minds become convinced that the virtual world we've created within our display is truly in 3 dimensions. You will know that your 3D efforts have succeeded if at some point you begin to think of your computer monitor or display NOT as a screen, but as a box or world with height, width, and depth!

Section 9.1 Rotating Objects in Space

The first program in this chapter is fairly simple. We will allow the placement of objects in the center of the graphics window and provide a method to rotate the object. The program will give the user the choice of which **glut** object to display based on the number keys. Also, the user will have the choice to display the **glut** object as wireframe or solid. Finally, lighting and perspective will be employed to give a richer 3D effect.

Save the following program as **glutshapes.py** or something similar. This is not an original program idea, but is based on a C program found in Richard Wright's fine "OpenGL SuperBible" text.[15]

```
# glutshapes.py
# Shapes and rotations with special keys

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys

import psyco
psyco.full()

#  Set the width and height of the window
global width
global height
```

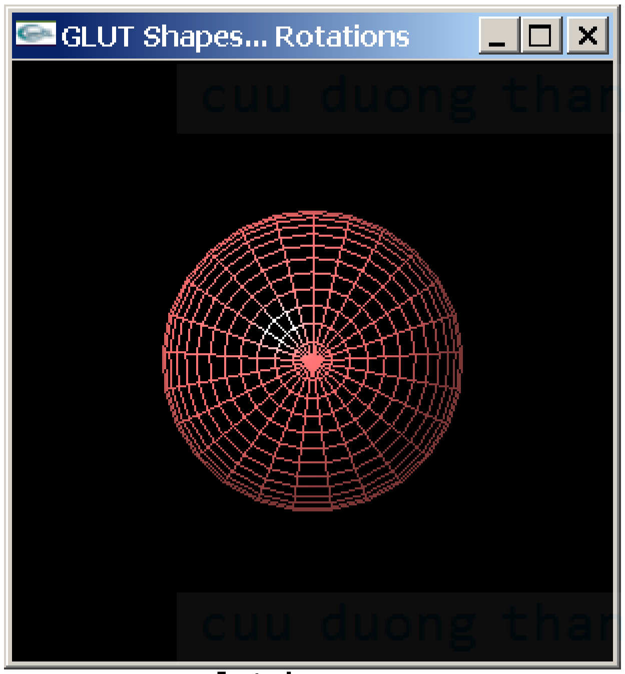
[15] Wright, Richard S. (1999). OpenGL SuperBible (2nd Edition). Pearson Education.

```
#  Global variables for rotation angles
global xrot
global yrot

xrot = 0.0
yrot = 0.0

width = 600
height = 600

# Light values and coordinates
global  ambientLight
global  diffuseLight
global  specular
global  specref

ambientLight =  (0.35, 0.35, 0.35, 1.0)
diffuseLight = ( 0.75, 0.75, 0.75, 0.7)
specular = (1.0, 1.0, 1.0, 1.0)
specref = (1.0, 1.0, 1.0, 1.0)

global glutshape
global solid
solid = "w"
glutshape = 1

def renderscene():
     global xrot
     global yrot

     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
     glPushMatrix()
     glRotatef(xrot, 1.0, 0.0, 0.0)
     glRotatef(yrot, 0.0, 1.0, 0.0)

     if solid == "w":
          if glutshape == 1:
               glutWireSphere(1.0, 25, 25)
          elif glutshape == 2:
               glutWireCube(1.0)
          elif glutshape == 3:
               glutWireCone(0.3, 1.1, 20, 20)
          elif glutshape == 4:
               glutWireTorus(0.3, 1.0, 10, 25)
          elif glutshape == 5:
               glutWireDodecahedron()
          elif glutshape == 6:
               glutWireOctahedron()
          elif glutshape == 7:
               glutWireTetrahedron()
          elif glutshape == 8:
               glutWireIcosahedron()
```

```
            elif glutshape == 9:
                 glutWireTeapot(1.0)
       elif solid == "s":
            if glutshape == 1:
                 glutSolidSphere(1.0, 25, 25)
            elif glutshape == 2:
                 glutSolidCube(1.0)
            elif glutshape == 3:
                 glutSolidCone(0.3, 1.1, 20, 20)
            elif glutshape == 4:
                 glutSolidTorus(0.3, 1.0, 10, 25)
            elif glutshape == 5:
                 glutSolidDodecahedron()
            elif glutshape == 6:
                 glutSolidOctahedron()
            elif glutshape == 7:
                 glutSolidTetrahedron()
            elif glutshape == 8:
                 glutSolidIcosahedron()
            elif glutshape == 9:
                 glutSolidTeapot(1.0)

       glPopMatrix()
       glutSwapBuffers()

def init():
    global width
    global height

    glClearColor(0.0, 0.0, 0.0, 1.0)

    # Enable depth testing
    glEnable(GL_DEPTH_TEST)

    glEnable(GL_LIGHTING)
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight)
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight)
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular)
    glEnable(GL_LIGHT0)
    glEnable(GL_COLOR_MATERIAL)
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE)
    glMaterialfv(GL_FRONT, GL_SPECULAR, specref)
    glMateriali(GL_FRONT, GL_SHININESS, 128)

    glColor3ub(230,100,100)

def specialkeys(  key,  x,  y):
       global xrot
       global yrot

       if key == GLUT_KEY_UP:
            xrot -= 2.0
```

```
      if key == GLUT_KEY_DOWN:
            xrot += 2.0
      if key == GLUT_KEY_LEFT:
            yrot -= 2.0
      if key == GLUT_KEY_RIGHT:
            yrot += 2.0

      glutPostRedisplay()

def reshape(  w,  h):
      lightPos = (-50.0, 50.0, 100.0, 1.0)
      nRange = 2.0

      if h==0:
            h = 1

      glViewport(0, 0, w, h)

      glMatrixMode(GL_PROJECTION)
      glLoadIdentity()

      if w <= h:
            glOrtho(-nRange, nRange, -nRange*h/w, nRange*h/w, -
nRange, nRange)
      else:
            glOrtho(-nRange*w/h, nRange*w/h, -nRange, nRange, -
nRange, nRange)

      glMatrixMode(GL_MODELVIEW)
      glLoadIdentity()

      glLightfv(GL_LIGHT0, GL_POSITION, lightPos)

def keyboard(key, x, y):
      global glutshape, solid
      if key == chr(27) or key == "q":
            sys.exit()
      try:
            if int(key) < 10:
                  glutshape = int(key)
      except:
            pass

      if key == "w" or key == "s":
            solid = key

      glutPostRedisplay()

def main():

    global width
    global height
    #  Setup for double-buffered display and depth testing
    glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE|GLUT_DEPTH)
    glutInitWindowPosition(100,100)
    glutInitWindowSize(width,height)
    glutInit(sys.argv)
    glutCreateWindow("GLUT Shapes... Rotations")

    init()

    glutReshapeFunc(reshape)

    glutDisplayFunc(renderscene)
    glutKeyboardFunc(keyboard)
    glutSpecialFunc(specialkeys)
    glutMainLoop()

main()
```

If everything runs correctly, you should see the screen in figure **`glutshape`** when the program begins.



`glutshape`

Pressing the number keys (1-9) should give you a unique shape for each number. Also, the "w" and "s" keys should toggle from wireframe to solid shapes. The arrow keys should allow you to rotate the shape on each axis. We now know how to rotate objects!