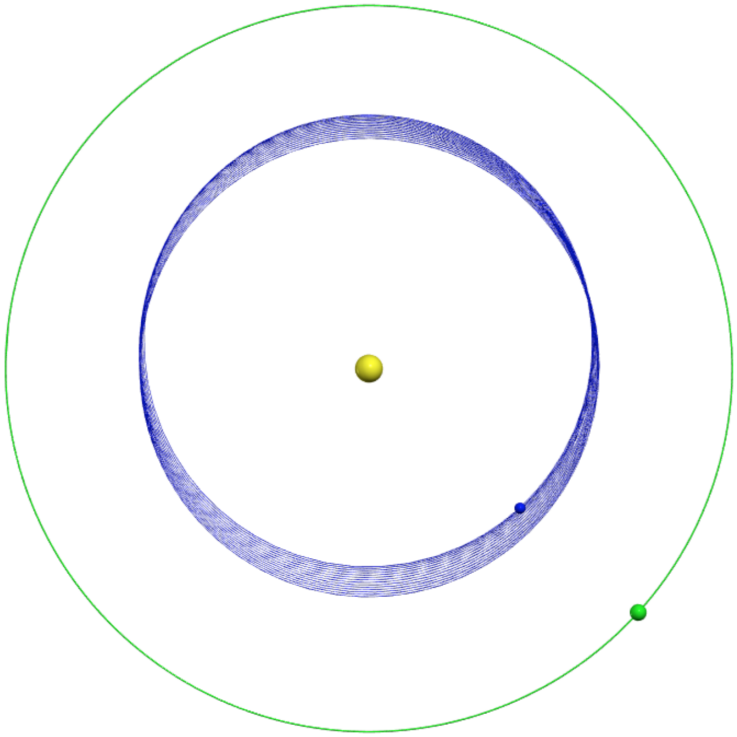
# Physics Simulations in Python

## A Lab Manual



Daniel V. Schroeder
Physics Department
Weber State University

May 2018

You can obtain the latest version of this manual at `http://physics.weber.edu/schroeder/scicomp/`. There you can also find the LaTeX source and figure files, to facilitate adapting this manual to different needs.

# Contents

Preface . . . . . iv

Project 1: Making Shapes . . . . . 1

Project 2: Projectile Motion . . . . . 11

Project 3: Pendulum . . . . . 25

Project 4: Orbits . . . . . 35

Project 5: Molecular Dynamics . . . . . 47

Project 6: Random Processes . . . . . 61

Project 7: Final Project . . . . . 73

# Preface

Introductory physics courses are full of simplifications: projectiles fly without air resistance, pendulums swing only at small angles, orbits are always circular, and no more than two particles move at any time. These kinds of simplifications are necessary and appropriate when you're first trying to understand the basic laws of nature. But the real world is far more complex, and far more interesting. Because the ultimate goal of physics is to understand the real world, students deserve a course that applies the laws of physics to more complex situations.

Fortunately, modern electronic computers make it possible to perform extremely lengthy calculations in a negligible amount of time. These days, therefore, computers offer the best avenue toward applying the basic laws of nature to complex and realistic physical systems. A computer program that models the behavior of a physical system is called a *computer simulation*. Creating and using computer simulations is an integral part of modern science and engineering.

This manual is intended for a hands-on introductory course in computer simulations of physical systems, using the Python programming language. The goals of the course are as follows:

- Learn enough of the Python language and the VPython and matplotlib graphics packages to write programs that do numerical calculations with graphical output;
- Learn some step-by-step procedures for doing mathematical calculations (such as solving differential equations) on a computer;
- Gain a better understanding of Newton's laws and other physical principles;
- Study a variety of physical systems that are too complex for simple pencil-and-paper calculations, and see what sorts of behavior emerge in such systems.

## Prerequisites

Before working through the projects in this manual you should have completed a semester of introductory physics, covering Newton's laws of motion, conservation principles, and a bit of thermodynamics. You should also have taken at least one semester of calculus. Prior expertise in writing computer programs is *not* required, but you should be fairly comfortable using a web browser, word processor, and spreadsheet, and you should have some experience at being careful with computer syntax (in any programming language).

## Required materials

Naturally, you'll need a computer. The first five projects use a cloud-based version of Python called GlowScript, so for those you can use any computer with an internet connection and a modern web browser. (A tablet device without a physical keyboard is not adequate.) For Project 6, you may need to install a free version of the Python language and environment (if you're not using a computer on which it is installed already).

Your GlowScript programs will be automatically saved on Google's servers, but for any other files you'll need to use either some other type of cloud storage or a USB memory stick for backup.

A pocket calculator (perhaps on your mobile phone) will sometimes come in handy.

Finally, you'll need a few low-tech materials such as scratch paper, pencils, and a small three-ring binder to hold this manual.

## How to use this manual

This manual is divided into six main chapters, corresponding to six separate projects. In each project you will write a computer program or (more often) a small number of closely related computer programs. Rather than giving you complete programs to run, the project instructions will provide only code fragments and general guidelines on how to write your programs. This way, once you have completed each program, it will be yours.

As you create your computer programs, you will inevitably have questions and encounter difficulties. While you should try to think things through for yourself whenever possible, don't spend too much time being stuck and getting frustrated. Ask your instructor or your lab partner or your other classmates for help. This is not a test.

Exercises and questions will be sprinkled among the instructions in this manual, with space for you to write your answers. Please make every effort work each exercise and answer each question immediately, before you read on.

The general premise of this manual is that you'll learn more by *trying* something than by reading a comprehensive *explanation* of it. Computer languages are like ordinary languages in this respect: We normally learn new words by hearing, reading, and using them in context, not by studying a dictionary. But if you want to see a term clearly defined, feel free to ask your instructor or look it up online.

Computer programming is fun because it's so open-ended. You'll constantly think of things to try that go beyond the explicit instructions. By all means, try anything you want! If you're not sure how to add a certain feature to one of your simulations, or if you're not sure whether it's practical to do so within a limited amount of time, be sure to ask your instructor.

When you finish a project, gather the instruction pages and staple them together with any printed output from your programs. This stapled packet, together with

the source code of your computer programs, will be your "lab report."

## What this manual is not

This manual is not a comprehensive introduction to the Python programming language. Many features of the language are not needed for the types of simulations we'll be doing, so we'll ignore them. Several other features will be used once or twice but never fully explained.

Standard distributions of Python come with dozens of packages (libraries) for carrying out a wide variety of common tasks. This manual will describe only a tiny fraction of them.

At some point you might want to distribute your "finished" VPython programs as web apps or stand-alone applications. This manual won't tell you how to do that.

I've tried to design the examples in this manual to illustrate good programming practices that are appropriate to the relatively small scale of the projects. This is not a treatise on the principles of professional software design.

This manual is not a textbook on numerical analysis, nor is it a reference work on numerical algorithms. We'll try out just a few algorithms, make some crude comparisons, and leave it at that.

The projects in this manual touch on some fascinating fields of physics, including nonlinear dynamics, celestial mechanics, and phase transformations. But this is not a textbook on any of these subjects.

Perhaps most importantly, this manual is not intended to be of any use whatsoever to someone who merely reads it without actually working through all of the projects and exercises.

## Why Python, VPython, and GlowScript?

Choosing a computer programming language always involves trade-offs. Fortunately, there are more choices today than ever before.

An obvious choice for this course would be one of the traditional computer languages like Fortran, C, or C++. These languages are widely used for scientific computation due to their flexibility and speed. The languages are defined by standards committees rather than by commercial vendors, and free versions are available. However, they have grown somewhat complex over the years, as features have been added while maintaining compatibility with older versions. Another disadvantage is that none of these languages include built-in support for graphics, and add-on graphics libraries tend to be difficult to install and use.

The Basic programming language was specifically designed to be easy to learn, and current versions of Basic have kept this feature. Because Basic is widely used by students and hobbyists, all modern versions include built-in, easy-to-use graphics support. Some versions (True Basic and Xojo) are cross-platform, but the most widely used version, Microsoft's Visual Basic, runs only on the Windows operating

system. The fragmentation of Basic into multiple versions, each with its own idiosyncracies, is a major disadvantage. Programs written in Basic also tend to run rather slowly. Most versions of Basic are sold commercially, though the prices are generally reasonable.

For mathematical calculations, the most convenient choice is often a specialized mathematical programming environment such as Mathematica (which I use a great deal), Maple, or Matlab. These packages contain sophisticated, speedy, built-in routines for a great variety of mathematical tasks, but their high overhead can make them rather slow and awkward when you need to program a custom step-by-step algorithm. Because they are commercial products aimed at relatively narrow markets, these packages tend to be expensive. (However, there is a free product called Octave that is very similar to Matlab.)

An earlier version of this manual used the Java programming language, introduced by Sun Microsystems (now Oracle) in 1995. Although based on C and C++, Java is easier to learn and use, and comes with standard cross-platform libraries for graphics and other common tasks. Its computational performance is remarkably good, though it isn't as fast as C or C++ or Fortran. But Java never really caught on with scientists, and its early use for web-delivered "applets" has now become obsolete. More importantly for us, programming in Java requires some inconvenient software installations and learning to use some rather advanced object-oriented features that are really superfluous in a first course in scientific computing.

For web-delivered applications, Java has now been replaced by JavaScript, a rather different language that was deliberately named to emphasize their superficial similarities. Every modern web browser can run JavaScript programs, and you're running them constantly as you surf the web. Writing JavaScript programs is a natural extension of creating ordinary web pages. Moreover, in recent years, the computational performance of the JavaScript engines in the most widely used web browsers has nearly matched that of Java, which in turn isn't far behind C (etc.). The main disadvantage of JavaScript is that for practical purposes it runs *only* in a web browser, so for security reasons it cannot access your computer's file system. This restriction has limited its use by scientists, at least for serious computational work.

Python is a relatively new, free, cross-platform language that scientists are using more and more widely. It is a simple language to get started with, and developers are creating a growing assortment of add-on packages to make various difficult tasks fairly easy. These add-on packages include several for numerical calculations and scientific graphics. One big disadvantage of Python is that every Python installation is a little different, depending on which Python version and add-on packages are present. Getting someone else's Python program to run on your Python system can therefore be a frustrating task. Another disadvantage is that most Python interpreters do not produce very efficient machine code, so Python programs tend to run rather slowly—necessitating the use of add-on packages for heavy-duty computation. Finally, a disadvantage for this course is that none of the graphics packages

included in the more common Python installations are especially convenient for creating animated graphics or interactive user controls.

The VPython (short for Visual Python) package is an attempt to address this last deficiency. It provides a very easy interface to a 3D graphics library, along with some auxiliary functions for handling vectors and animation. It was created specifically for use in undergraduate physics courses, and it is being maintained and improved by Bruce Sherwood, a physics teacher and textbook author. Unfortunately, the VPython package has never been a standard part of most Python installations, and its graphics systems have not always worked well with all Python environments. The difficulty of installing VPython and getting it to work correctly has therefore been a barrier to its use.

More recently, though, Sherwood and others have created GlowScript: a cloud-based environment for writing and running 3D graphics programs in a web browser. Originally GlowScript required programming in JavaScript, but now it has a built-in facility for translating Python code into JavaScript behind the scenes, so to the programmer it appears very similar to VPython. GlowScript-VPython therefore offers most of the advantages of VPython, without any of the installation hassles. And it runs significantly faster than Python in most cases, because the JavaScript engines in modern web browsers are so good. The disadvantages of this environment are mostly the same as those of JavaScript: A GlowScript program cannot directly access your computer's file system, and (for the same reason) it does not have access to the vast world of Python add-on packages. (The common math functions and a few other essential functions from Python packages are, however, incorporated into GlowScript.)

The bottom line for this manual is that we will use GlowScript-VPython for Projects 1 through 5. In Project 6, however, we will switch to a more standard Python installation, in order to give you some experience with that environment.

## References

Although the project instructions in this manual are fairly self-contained, you may wish to consult the following references for more information on the Python language, the GlowScript-VPython environment, numerical computation, and physics simulations.

- VPython online documentation, available via the "Help" link from the GlowScript environment or at `http://www.glowscript.org/docs/VPythonDocs/index.html`. Although we won't use every feature described, much of this reference material will be essential reading. Fortunately, it's concise and well written.

- *Python for Non-Programmers* is a web page with numerous links to Python tutorials and other resources for beginners: `https://wiki.python.org/moin/BeginnersGuide/NonProgrammers`.

- *University Physics Volumes 1 and 2* (OpenStax, 2016), `https://openstax.org/details/books/university-physics-volume-1` and `https://openstax.org/details/books/university-physics-volume-2`. If you need to refresh your memory of the definitions and principles from your introductory physics course, the free OpenStax textbooks are convenient references (although any other textbook from such a course will also do). Volume 1 covers topics in mechanics, while Volume 2 includes thermodynamics.

- Mark Newman, *Computational Physics*, revised and expanded edition (CreateSpace, 2013). This is a reasonably comprehensive (and reasonably priced) textbook on numerical methods, using the Python language (including the matplotlib and VPython graphics packages) and written with physics students in mind.

- Alejandro L. Garcia, *Numerical Methods for Physics*, second edition, Python version (CreateSpace, 2017). This well-written textbook was originally published by Prentice Hall, and used the Matlab and C++ languages. A very affordable version became available through CreateSpace a few years ago, and the brand new Python version should be a strong competitor to Newman's book.

- Jesse M. Kinder and Philip Nelson, *A Student's Guide to Python for Physical Modeling* (Princeton University Press, 2015). This rather slim book is just what it says: an introduction to the Python language intended for science students. It uses the packages of a standard Python installation (no VPython), and discusses numerical analysis topics only briefly.

- Harvey Gould, Jan Tobochnik, and Wolfgang Christian, *An Introduction to Computer Simulation Methods*, third edition (CreateSpace, 2017). An innovative textbook that covers far-ranging physics applications, mostly at a level accessible to undergraduates. This book inspired much of the manual you're now reading, and provides a wealth of ideas for further projects. Earlier editions of this book used True Basic, and the third edition uses Java, but I've always ignored most of the code and focused on the ideas and algorithms. Originally published by Addison-Wesley, this book is now much more affordable and you can even download a free electronic version at `https://www.compadre.org/osp/items/detail.cfm?ID=7375`.

- Nicholas J. Giordano and Hisao Nakanishi, *Computational Physics*, second edition (Prentice Hall, 2006). This book is remarkably similar in outline and level to Gould and Tobochnik, and also used the True Basic language in its first edition. The second edition gives algorithms in pseudo-code, with an accompanying web site that provides implementations in True Basic and Fortran. This book is more focused than Gould and Tobochnik, with fewer topics but more discussion of the results of the simulations. I've borrowed quite a few ideas from it while writing this manual, and I recommend it as another

source of ideas for further projects. If it weren't so expensive I might have assigned it as a textbook for this course.

- William H. Press et al., *Numerical Recipes*, third edition (Cambridge University Press, 2007), `http://numerical.recipes/`. By far the most widely used reference on numerical algorithms, aimed at professional researchers and graduate students. Well written but quite advanced. Code implementations are in C++, although earlier editions are also available in C and Fortran versions. You probably won't get any use out of this book during this course, but if you go on in computational science you'll eventually need a copy (or an electronic subscription).

- Ian R. Gatland, "Numerical integration of Newton's equations including velocity-dependent forces," *American Journal of Physics* **62**, 259–265 (1993), `https://doi.org/10.1119/1.17610`. This excellent article emphasizes the advantages of the Euler-Richardson algorithm and explains how to implement adaptive step-size control for this algorithm. Includes references to earlier *AJP* articles that advocate other simple algorithms for Newton's equations.

Physics 2300 Name _____

Lab partner _____

# Project 1: Making Shapes

## Saying *hello*

The first goal in learning any new computer programming environment is always the same: Write and run a program to print (or display) a brief message, traditionally "Hello, world!"

Why bother with such a boring program? Because the steps required to write and run even the most trivial program can be quite intricate. For some programming environments you may need to install software, configure the software to work with your computer's directory system, and then learn to use various software tools for editing, compiling, linking, and launching your program. Then you need to learn enough about the programming language, and about the associated software libraries for producing the type of output you want, in order to type in the code needed to produce that output. The number of things that can go wrong during this whole process is enormous.

Fortunately, the GlowScript-VPython system is one of the easiest of all possible programming environments for getting started. There are still some things that can go wrong, but even if they do, I don't think they'll take long to resolve. Here are the steps:

1. Launch a web browser (Google Chrome is recommended, but most others should work), and go to the site `glowscript.org`.

2. Sign in with a Google account. Your WSU email credentials should work, or you can use a personal Google account.

3. Follow the link where it says "your programs are **here**."

4. Click on the "Create New Program" link.

5. In the dialog box that appears, type the title "MakingShapes", then click the "Create" button.

6. You will now see an editing space that is blank except for the line "GlowScript 2.7 VPython" (possibly with a different version number). Click in the white space below that line and type the following, verbatim:

```
print("Hello, GlowScript-VPython!")
```

   (Actually you can put almost any message you like between the quotes.)

7. Then click the "Run this program" link at the top of the page. Your editing space should then vanish, replaced by a new text area with the words "Hello, GlowScript-VPython!" (or whatever message you put in your code).

Well, did it work? If so, congratulations! You're up and running. If not, don't worry; if you can't resolve the problem yourself in a few minutes, your instructor, or perhaps a classmate, can probably help. (The most likely stumbling block is with the Google account sign-in. If that went smoothly but your program doesn't work, be sure to double-check your spelling, capitalization, and punctuation.)

GlowScript-VPython is a cloud-based system that stores your programs on Google's servers. The interface is bare-bones, and I hope you won't have any trouble seeing how to create new programs, copy, rename, and delete them, and organize them into folders. If you have any questions about these procedures, be sure to ask your instructor.

Now let's look at your two-line program in a bit of detail. Click the "Edit this program" link in order to see your code again. The first line, which you don't even need to type, tells the GlowScript system what language (VPython) and version (2.7) you want to use. This line is required, and for this course you should always leave it alone.

We say that the next line, which you typed, "calls the `print` function and passes it a string of text characters." Sorry about the jargon words *call*, *function*, *pass*, and *string*. If you're an experienced programmer then these words are probably already in your vocabulary; if this is your first time writing code, then you'll need to pay attention to these words and practice using them correctly. For now, please notice two aspects of Python syntax:

- The *string* of text is enclosed in double-quote marks. Single-quote marks would also have worked, as long as the beginning and ending quotes match.
- The information *passed* to the function (its *parameter*) is enclosed in parentheses. In this case there is a single parameter (a string), but we'll soon see examples of functions that take multiple parameters, separated by commas.

The idea of a *function* is to hide the details of what's happening from your code (or at least from this portion of it), so all you need to know is the function name, what parameter(s) to pass, and what the function does—but not how it works. The `print` function has to do an awful lot to make those words appear on your screen, but you needn't know the details.

## Your first shape

Next, on a new line below your `print` instruction, type the following simple instruction:

```
box()
```

Here you're calling a function called `box`, and passing it no parameters at all (but notice that the parentheses are still required). Run the program again, and you should see a black rectangular area (called a *canvas*) containing a light gray square (the box). Again, this function is doing a great deal of work, but you needn't worry about how.

The box that you see lives in an imaginary *three*-dimensional space, but you're viewing it from one side, at relatively close range. To change the perspective, you can do three things:

- **Rotate:** Press the right mouse button and drag one way or another, or, if you don't have a right mouse button, press the *control* key and drag using your mouse or trackpad.
- **Zoom:** Use the scroll wheel on your mouse, or, if there isn't one, press the *alt* or *option* key and drag using your mouse or trackpad. (On some trackpads you can also drag using two fingers.)
- **Pan:** Hold down the *shift* key while you drag using the mouse or trackpad.

I hope you're impressed by how hard that little `box` function is working!

You can change the attributes of your box by passing some parameters to the `box` function. Try this:

```
box(pos=vector(1,0,0), size=vector(.5,.3,.2), color=color.red)
```

Here you're providing three parameters, separated by commas, and you're identifying them by their names (a neat feature of Python), which means you can provide them in any order. The `pos` parameter specifies the position of the center of the box; the `size` parameter specifies its dimensions; and the `color` parameter is self-explanatory. In the first two cases, the parameter values are three-dimensional vectors, which you create using the `vector` function. This function in turn takes three parameters, `x`, `y`, and `z`, which you needn't name as long as you provide them in that order. Initially (before you rotate the scene), the $x$ direction points to the right; the $y$ direction points up; and the $z$ direction points directly outward, toward you (or toward the "camera").

To learn more about colors, click the **Help** link at the upper-right corner of the window. (I suggest opening the help page in a new browser window or tab.) Then, from the second drop-down menu in the left sidebar, choose "Color/Opacity".

There you'll find a list of pre-defined colors, and also see how you can use the `vector` function to create arbitrary colors.

**Exercise:** Although VPython includes a predefined `color.purple`, it isn't very vivid. Figure out how to make your box a brighter (more saturated) shade of purple, and write down how you did it here:

**Exercise:** Create at least two more boxes, so you'll have a total of at least three, each with different positions, sizes, and colors. Keep their positions within the range $-5$ to 5 in each dimension, and keep their sizes small enough to leave plenty of room for more shapes within that range.

### More shapes

VPython provides functions for creating quite a variety of shapes, but in this course you'll need just two others: spheres and cylinders. Try this instruction to create a sphere:

```
sphere(radius=0.25)
```

The `sphere` function can also accept the `pos` and `color` parameters, so use those now to change the defaults according to your taste. Don't forget to separate the parameters by commas!

**Question:** What happens if you omit the `radius` parameter when you call the `sphere` function?

**Exercise:** Create a second sphere, again within the range $-5$ to 5 in each dimension, with a (reasonably small) radius and color of your choosing.

Now try this instruction to create a cylinder:

```
cylinder(axis=vector(0,1.5,0))
```

Here the `axis` parameter is a displacement vector that takes you from one end of the cylinder to the other. You can also provide the `pos`, `radius`, and `color` parameters, so please put in all three at this time, to change the defaults to suit your taste.

If you check carefully, you'll discover that the `pos` of a cylinder is located at one of its ends, rather than in its center as for a sphere or a box.

**Exercise:** It would be helpful if your 3D space had some coordinate axes, right? So create some now, in the form of very skinny cylinders running from $-5$ to 5 in each of the three dimensions. Color them light gray.

## Errors and debugging

You may have already had to deal with some error messages, alerting you to typographical errors (sometimes called *bugs*) in your program. Now let's take a deliberate look at one of these messages, and what you should do about it.

**Exercise:** Remove one of the commas that separate the parameters in one of your `box`, `sphere`, or `cylinder` function calls. Then try to run your program, and write down the error message that appears (or at least the beginning of it, including the line number). Click the "Edit this program" link to go back to your code. Is the line number in the message the same as the line in which you introduced the error?

In my own experience, the line number is usually the most useful part of an error message. The rest of the message is often unhelpful, or at least unnecessary. But even the line number can be off by a little, so when you're looking for the error in the code, be sure to look at the surrounding lines as well. Some GlowScript error messages don't even include line numbers, in which case you should carefully scrutinize whatever changes you made most recently to your code.

Besides missing commas, some other common types of Python errors are other incorrect punctuation, incorrect spelling, and inconsistent capitalization (since all words in Python are case-sensitive).

## Variables and arithmetic

The various shape attributes that you've been setting—`pos`, `radius`, `color`, and so on—are examples of what we call *variables*. A variable, in computer programming, is a named location in the computer's memory in which some information can be stored. The equals sign tells the computer to store the information on its right in the variable whose name is on its left.

Besides these pre-named variables, you can create your own. The next exercise demonstrates this.

**Exercise:** To keep track of your shapes, you can give them variable names. Name your $x$ axis by inserting "`xaxis = `" at the beginning of the line, like this:

```
xaxis = cylinder( . . . )
```

Similarly, name the other two axes `yaxis` and `zaxis`.

These variable names make your code easier to read, even if you never use them in any other way. But here's a way to make use of one of them:

**Exercise:** Your three axes probably all have the same radius and the same color. So instead of writing out the same specifications three times, put them into only the instruction that creates the $x$ axis, and then in the other two, say this:

```
radius=xaxis.radius, color=xaxis.color
```

Now if you decide to change the radius or the color, you'll need to make the change in only one place (try it!).

**Exercise:** Create a dumbbell shape, by combining a cylinder with two spheres. Make the cylinder first, with any suitable attributes, naming it `bar`. Then, when you create the two spheres, simply set the position of one of them to `bar.pos` and the position of the other to `bar.pos + bar.axis`. This is an example of how you can use the `+` sign to do `vector` addition. Also set the radius of each sphere to `bar.radius*3`, and set the color of each sphere to `bar.color`.

The previous exercise included your first examples of addition and multiplication. Python also provides the `-` operator for subtraction and the `/` operator for division, as well as other operators that you'll learn later. You'll need to do some subtraction in the next exercise.

**Exercise:** Create a small table (the furniture kind) in your simulated 3D space, consisting of a box for the top plus four cylinders for the legs, oriented with the $y$ direction up. For convenience and flexibility, start by setting the values of some variables:

```
tablex = 2.5
tabley = -2
tablez = -1
```

Don't feel obligated to use the same numbers as these; I'm just suggesting some variable names and showing how to assign values to them. Similarly, introduce and set the variables `tableLength`, `tableWidth`, `tableHeight`, `legRadius`, and `tableColor`. Instead of providing an explicit number for `legRadius`, use multiplication or division to make it a small fraction of `tableWidth`. Then create the box that will be the table's top, centering it at `vector(tablex, tabley, tablez)` and setting its length and width to the corresponding variables. The height of the table's *top*, however, should be only a small fraction of its total height. (If the line of code to create the tabletop gets too long, you can break it into two; for readability you should then indent the second line by a couple of tabs.) Finally, create the leg cylinders, using arithmetic to position them near the table's four corners no matter what the values of all your variables are set to. Try changing some of these values to make sure your table still looks like a table for any reasonable values.

**Exercise:** Here's something easier. The "canvas" in which your shapes appear has its own variable name: `scene`. This variable, like `color` and `xaxis`, is a so-called *object* with its own attributes. (An attribute is essentially a sub-variable that is associated with some larger, more inclusive variable.) One of the attributes of

`scene` is the background color, which is black by default. Change it to white by inserting this line:

```
scene.background = color.white
```

Check that this works, then change the background color to a pale sky blue, or to some other very light color that won't use a lot of toner when you later print your scene. Also set the variable `scene.range` to 5; this will set the initial zoom level to put $y = 5$ at the top edge of the canvas and $y = -5$ at the bottom edge.

## Comments

By now your program is getting long enough to require some organization—not for the computer's sake, but for your own as you look at the code.

**Exercise:** Divide your lines of code into logical groups, separated by blank lines. Then, at the top of each group, insert an introductory *comment* such as

```
# Create a table out of a box and four cylinders:
```

A comment in Python begins with the `#` character and continues until the end of the line. To create a multi-line comment, you need to put the `#` character at the beginning of each line. The computer completely ignores comments when it runs your program.

**Exercise:** Put a multi-line comment at the top of your program, right after the line "GlowScript 2.7 VPython", to indicate the name of your program, your own name, and the date when you created it, and to give a one-sentence description of what it does. (From now on, please include a similar comment at the top of every program that you write.)

## Animation

In this course you'll need to depict not just static objects but also physical processes that play out in time. That calls for *animation*, and one of the advantages of VPython is that it makes animation extremely easy.

**Exercise:** Create a small box near $x = -5$ and call it `movingBox`. Then insert the following code and see what it does:

```
# Move a box across the scene in a straight line:
while movingBox.pos.x < 5:
    rate(50)
    movingBox.pos.x += 0.05
```

Be sure to indent the last two lines; you can use the tab key for this.

If this code works, you should see the box move smoothly across the scene, stopping when it reaches $x = 5$. The code that accomplishes this magic is called a

*loop*, which in general means a block of code that the computer executes repeatedly. Some languages use curly braces to denote which lines of code are part of the loop, but in Python this is indicated by indentation. This particular type of loop begins with the special word `while`, followed by the *condition* that must be true as long as the loop continues, and then followed by a colon. The condition can involve any of the comparison operators `<`, `>`, `<=`, `>=`, `==` (for "is equal to"), and `!=` ("is not equal to"), as well as the "boolean" operators `and`, `or`, and `not`.

The two indented lines themselves also require explanation. The first of them, `rate(50)`, tells the computer how fast to try to execute the loop—in this case, 50 times per second. The next line contains the two-character operator `+=`, which tells the computer to add the quantity on the right *onto* the variable on the left. Saying `x += 42` is completely equivalent to saying `x = x + 42`. (When appropriate, you can similarly say `-=`, `*=`, and `/=`.)

**Exercise:** Change the parameter `50` in the `rate` function to some other number, and make sure this change has the expected effect.

**Exercise:** Add a line or two to your code to make the box move along a diagonal, rather than directly from left to right. Make sure the box doesn't move too far in some other direction before the motion stops.

**Exercise:** Add code to your program to create a small sphere and then move that sphere at a steady speed once around a circle in the $xy$ plane, centered at the origin. Use a variable called `theta` for the angle around the circle, and set this variable equal to zero before your `while` loop begins. Also use a variable called `r` for the circle's radius, and `x` and `y` for its rectangular coordinates. To calculate `x` and `y` you can use the built-in trigonometric functions `cos` and `sin`, for example, `x = r * cos(theta)`. Be careful, though, because the trig functions assume that the parameter you pass to them is in *radians*; keep this in mind when deciding how much to change `theta` during each loop iteration, and in deciding what conditions to use in the `while` statement. To actually move the sphere in the graphics scene, use an instruction of the form `movingSphere.pos = vector(x, y, 0)`.

**Exercise:** Look in the VPython documentation (via the Help link, if you don't already have it open) for instructions on how to "attach a trail" to an object as it moves. (Use the easier `make_trail` parameter, not the more complicated `attach_trail` function.) Attach a trail of points to your moving sphere, using the `interval` attribute to space them a little farther apart than the default.

**Exercise:** Remember the `print` function? Insert a couple more calls to it in your code, to print out suitable messages when each of your animation loops (one for the box, one for the sphere) has finished.

## Graphing

Often in science we want to visualize a phenomenon not in physical space but instead in a "space" of other variables such as time, velocity, and so on. While we `could` use a 3D VPython canvas to make such an abstract graph, VPython also provides a `graph` object that's usually more suitable.

As an example, let's graph the $x$ and $y$ positions of your moving sphere as functions of "time". Here is the code to set up the graph, with two data sets that will be plotted as dots in different colors:

```
graph(width=400, height=250)
xDots = gdots(color=color.green)
yDots = gdots(color=color.magenta)
```

(I've set the `width` and `height` attributes to make the graph a little smaller than the default, so it will fit more easily on small screens and printed pages.) After this initial setup, you add a dot to the graph by saying something like

```
xDots.plot(t,x)
```

and similarly for `yDots`.

**Exercise:** Insert the code to set up a graph just before your existing code to move the sphere around in a circle. Also, in that existing code, insert code to create a variable `t` to represent time, initially equal to zero and increasing by 1 during each loop iteration. Then, also inside the `while` loop, insert the lines that plot each $(t, x)$ and $(t, y)$ pair, in green and magenta, respectively. Run the program to make sure it all works.

**Exercise:** Look up the other attributes of the `graph` object in the VPython documentation, then set the graph's background color to white, label both axes appropriately, and give it a suitable title. Notice in the documentation that you can also specify the ranges of values for the graph to show—but for your current graph, leave these unspecified to enable "auto-scaling".

## Finishing up

Congratulations! You now know how to use GlowScript VPython to do basic arithmetic, draw and animate shapes, create graphs, and produce text output. You'll get plenty of practice with all of these tasks in later projects. For now, just answer a few more questions and then you'll be finished with this assignment.

**Exercise:** At the top of the next page, describe one specific bug (error) that you had to fix while working on this project, or, if you prefer, one specific point at which you became frustrated or confused. If you can't think of an example of either, then browse the VPython documentation and describe one interesting feature that you haven't used in this project.

**Exercise:** When you type your code in the GlowScript editing window, it automatically tries to use colors to distinguish different categories of words and symbols. List these colors below and describe what each seems to be for, keeping in mind that the process is rather buggy so some of the coloring can be inconsistent.

**Exercise:** List the names of all the VPython functions you used in this project. (Hint: Look in your code for parentheses.)

Before turning in this assignment, please look over your program and make sure it is well organized and easy to read. The code should be divided up into logically distinct groups, separated by blank lines, each introduced by a helpful comment.

Finally, run your program one last time and use your browser's print command to print the window contents, showing your 3D scene (with a light-colored background!) with the graph and text output below it. Make sure that all of this fits on a single printed page. Staple that page to the back of these instruction pages and turn in the packet to your instructor.

To turn in your actual code, you have two choices. The easier choice is to copy it into a public GlowScript folder and email a link to the program (or the folder) to your instructor. However, if for privacy reasons you do not wish to put your program into a public folder, you may also carefully copy and paste your code into an email message sent to your instructor, or copy and paste it into a plain text file and then email that file as an attachment (use the file extension .txt or .py). Please use "Physics 2300 Project 1" as the subject line of your email.

Physics 2300 Name _____

Lab partner _____

# Project 2: Projectile Motion

You now know enough about VPython to write your first simulation program.

The idea of a *simulation* is to program the laws of physics into the computer, and then let the computer calculate what happens as a function of time, step by step into the future. In this course those laws will usually be Newton's laws of motion, and our goal will be to predict the motion of one or more objects subject to various forces. Simulations let us do this for *any* forces and *any* initial conditions, even when no explicit formula for the motion exists.

In this project you'll simulate the motion of a projectile, first in one dimension and then in two dimensions. When the motion is purely vertical, the state of the projectile is defined by its position, $y$, and its velocity, $v_y$. These quantities are related by

$$v_y = \frac{dy}{dt} \approx \frac{\Delta y}{\Delta t} = \frac{y_{\text{final}} - y_{\text{initial}}}{\Delta t}. \tag{2.1}$$

In a computer simulation, we already know the current value of $y$ and want to predict the future value. So let's solve this equation for $y_{\text{final}}$:

$$y_{\text{final}} \approx y_{\text{initial}} + v_y \, \Delta t. \tag{2.2}$$

Similarly, we can predict the future value of $v_y$ if we know the current value as well as the acceleration:

$$v_{y,\text{final}} \approx v_{y,\text{initial}} + a_y \, \Delta t. \tag{2.3}$$

These equations are valid for any moving object. For a projectile moving near earth's surface without air resistance, $a_y = -g$ (taking the $+y$ direction to be upward). In general, $a_y$ is given by Newton's second law,

$$a_y = \frac{\sum F_y}{m}, \tag{2.4}$$

where $m$ is the object's mass and the various forces can depend on $y$, $v_y$, or both.

In a computer simulation of one-dimensional motion, the idea is to start with the state of the particle at $t = 0$, then use equations 2.2 through 2.4 to calculate $y$ and $v_y$ at $t = \Delta t$, then repeat the calculation for the next time interval, and the next, and so on. Fortunately, computers don't mind doing repetitive calculations.

But there's one remaining issue to address before we try to program these equations into a computer. Equation 2.2 is ambiguous regarding *which* value of $v_y$ appears on the right-hand side. Should we use the initial value, or the final value, or some intermediate value? In the limit $\Delta t \to 0$ it wouldn't matter, but for any

nonzero value of $\Delta t$, some choices give more accurate results than others. The easiest choice is to use the *initial* value of $v_y$, since we already know this value without any further computation. Similarly, the simplest choice in equation 2.3 is to use the initial value of $a_y$ on the right-hand side.

With these choices, we can use the following Python code to simulate projectile motion in one dimension without air resistance:

```
while y > 0:
    ay = -g
    y += vy * dt    # use old vy to calculate new y
    vy += ay * dt   # use old ay to calculate new vy
    t += dt
```

This simple procedure is called the *Euler algorithm*, after the mathematician Leonard Euler (pronounced "oiler"). As we'll see, it is only one of many algorithms that give correct results in the limit $\Delta t \to 0$.

**Exercise:** Write a VPython program called `Projectile1` to simulate the motion of a dropped ball moving only in the vertical dimension, using the Euler algorithm as written in the code fragment above. Represent the ball in the 3D graphics scene as a sphere with any reasonable radius (for display only—please ignore the radius in all physics calculations), and make a very shallow box at $y = 0$ to represent the ground. Use a light background color for eventual printing. The ball should leave a trail of dots as it moves. Be sure to put in the necessary code to initialize the variables (including `g`, putting all values in SI units), add a `rate` function inside the simulation loop, and update the ball's `pos` attribute during each loop iteration. Also be sure to format your code to make it easy to read, with appropriate comments. Use a time step (`dt`) of 0.1 second. Start the ball at time zero with a height of 10 meters and a velocity of zero. Notice that I've written the loop to terminate when the ball is no longer above $y = 0$. Test your program and make sure the animated motion looks reasonable.

**Exercise:** Most of the space on your graphics canvas is wasted. To fix this, set `scene.center` to half the ball's starting height (effectively pointing the "camera" at the middle of the trajectory), and set `scene.width` to 400 or less.

**Exercise:** Let's focus our attention on the *time* when the ball hits the ground, and on the *final velocity* upon impact. To see the numerical values of these quantities, add the following line to the end of your program:

```
print("Ball lands at t =", t, "seconds, with velocity", vy, "m/s")
```

Here we're passing five successive parameters to the `print` function: three quoted strings that are always the same (called *literal* strings), and the two variables whose values we want to see. If you look closely, you'll notice that the `print` function adds a space between successive items in the output.

**Exercise:** Are the printed values of the landing time and velocity what you would expect? Do a short calculation in the space below to show the expected values, as you would predict them in an introductory physics class.

**Exercise:** The time and velocity printed by your program do not apply at the instant when the ball reaches $y = 0$, because that instant occurs somewhere in the middle of the final time interval. Add some code after the end of the `while` loop to estimate the time when the ball actually reaches $y = 0$. (Hint: Use the final values of `y` and `vy` to make your estimate. The improved time estimate still won't be exact, but it will be much more accurate than what your program has been printing so far.) Have the program print out the improved value of $t$ instead, as well as an improved estimate of the velocity at this time. Write your new code and your new results in the space below. Please have your instructor *check* your answer to this exercise before you go on to the next.

**Exercise:** The inaccuracy caused by the nonzero size of `dt` is called *truncation error*. You can reduce the size of the truncation error by making `dt` smaller. Try it! (You'll want to adjust the parameter of the `rate` function to make the program run faster when `dt` is small. The slick way to do this is to make the parameter a *formula* that depends on `dt`. You'll also want to set the ball's `interval` attribute in a similar way, so the dot spacings are the same for any `dt`.) How small must `dt` be to give results that are accurate to four significant figures? Justify your answer.

## Air resistance

There's not much point in writing a computer simulation when you can calculate the exact answer so easily. So let's make the problem more difficult by adding some air resistance. At normal speeds, the force of air resistance is approximately proportional to the *square* of the projectile's velocity. This is because a faster projectile not only collides with *more* air molecules per unit time, but also imparts more momentum to each molecule it hits. So we can write the magnitude of the air force as

$$|\vec{F}_{\text{air}}| = c|\vec{v}|^2, \tag{2.5}$$

for some constant $c$ that will depend on the size and shape of the object and the density of the air. The *direction* of the air force is always directly opposite to the direction of $\vec{v}$ (at least for a symmetrical, nonspinning projectile).

**Exercise:** Assuming that the motion is purely in the $y$ direction, write down a formula for the $y$ component of the air force, in terms of $v_y$. Your formula should have the correct sign for *both* possible signs of $v_y$. (Hint: Use an absolute value function.) If you have any doubt about whether you've found the correct formula, have your instructor check it before you go on.

**Exercise:** Now modify your `Projectile1` program to include air resistance. Define a new variable called `drag`, equal to the coefficient $c$ in equation 2.5 divided by the ball's mass. Then add a term for air resistance to the line that calculates `ay`. Python's absolute value function is called `abs()`. Run the program for the following values of `drag`: 0 (to check that you get the same results as before), 0.01, 0.1, and 1.0. Use the same initial conditions as before, with a time step small enough to give about four significant figures. Write down the results for the time of flight and final speed below.

**Question:** What are the SI units of the `drag` constant in your program?

**Question:** How can you tell that your program is accurate to about four significant figures, when you no longer have an "exact" result to compare to?

**Exercise:** Modify your `Projectile1` program to plot a graph showing the ball's velocity as a function of time. (By default the graph will appear above or below the graphics canvas, and you may leave it there if you like. If you would prefer to place the graph to the right of the canvas, you can do so by creating the graph first, setting its attribute `align="right"`, and immediately plotting at least one point on the graph to make it appear before you set up the canvas.) Use the `xtitle` and `ytitle` attributes to label both axes of the graph appropriately, including the units of the plotted quantities. Use the `interval` parameter of the `gdots` function to avoid plotting a dot for every loop iteration (which would be pretty slow). Run your program again with `drag` equal to 1.0, and print the whole window including your canvas and graph. Discuss the results briefly.

**Exercise:** When the projectile is no longer accelerating, the forces acting on it must be in balance. Use this fact to calculate your projectile's terminal speed by hand, and compare to the result of your computer simulation.

## A better algorithm

Today's computers are fast enough that so far, you shouldn't have had to wait long for answers accurate to four signficant figures. Still, the Euler algorithm is sufficiently inaccurate that you've needed to use pretty small values of `dt`, making the calculation rather lengthy. Fortunately, it isn't hard to improve the Euler algorithm.

Remember that the Euler algorithm uses the values of $v_y$ and $a_y$ at the *beginning* of the time interval to estimate the changes in the position and velocity, respectively.

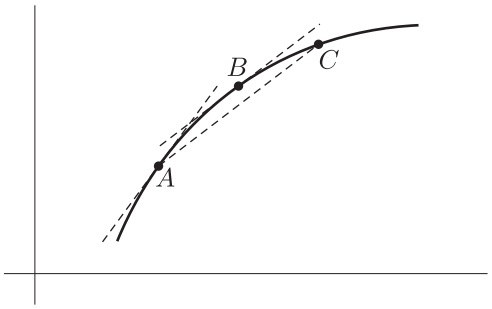



Figure 2.1: The derivative of a function at the middle of an interval (point $B$) is a much better approximation to the average slope ($AC$) than the derivative at the beginning of the interval (point $A$).

A much better approximation would be to instead use the values of $v_y$ and $a_y$ at the *middle* of the time interval (see Figure 2.1). Unfortunately, these values are not yet known. But even a rough estimate of these values should be better than none at all. Here is an improved algorithm that uses such a rough estimate:

1. Use the values of $v_y$ and $a_y$ at the beginning of the interval to estimate the position and velocity at the middle of the time interval.

2. Use the estimated position and velocity at the middle of the interval to calculate an estimated acceleration at the middle of the interval.

3. Use the estimated $v_y$ and $a_y$ at the middle of the interval to calculate the changes in $y$ and $v_y$ over the whole interval.

This procedure is called the *Euler-Richardson algorithm*, also known as the *second-order Runge-Kutta algorithm*.

Here is an implementation of the Euler-Richardson algorithm in Python for a projectile moving in one dimension, without air resistance:

```
while y > 0:
    ay = -g                     # ay at beginning of interval
    ymid = y + vy*0.5*dt        # y at middle of interval
    vymid = vy + ay*0.5*dt      # vy at middle of interval
    aymid = -g                  # ay at middle of interval
    y += vymid * dt
    vy += aymid * dt
    t += dt
```

The acceleration calculations in this example aren't very interesting, because $a_y$ doesn't depend on $y$ or $v_y$. Still, the basic idea is to estimate $y$, $v_y$, and $a_y$ in the middle of the interval and then use these values to update $y$ and $v_y$. Although each

step of the Euler-Richardson algorithm requires roughly twice as much calculation as the original Euler algorithm, it is usually many times more accurate and therefore allows us to use a much larger time interval.

**Exercise:** Write down the correct modifications to the lines that calculate `ay` and `aymid`, for a projectile falling *with* air resistance. (Be careful to use the correct velocity value when calculating `aymid`!)

**Question:** One of the lines in the Euler-Richardson implementation above is not needed, even when there's air resistance. Which line is it, and why do you think I included it if it isn't needed?

**Exercise:** Modify your `Projectile1` program to use the Euler-Richardson algorithm. For a `drag` constant of 0.1 and the same initial conditions as before ($y = 10$ m, $v_y = 0$), how small must you now make `dt` to get answers accurate to four significant figures? Write down the date to justify your answer. (You should find that `dt` can now be *significantly* larger than before. If this isn't what you find, there's probably an error in your implementation of the Euler-Richardson algorithm.)

Your `Projectile1` program is now finished. Please make sure that it contains plenty of comments and is well-enough formatted to be easily legible to human readers.

## Two-dimensional projectile motion

Simulating projectile motion is only slightly more difficult in two dimensions than in one. To do so you'll need an $x$ variable for every $y$ variable, and about twice as many lines of code to initialize these variables and update them within the simulation loop. To calculate the $x$ and $y$ components of the drag force, it's helpful to draw a picture (see Figure 2.2).

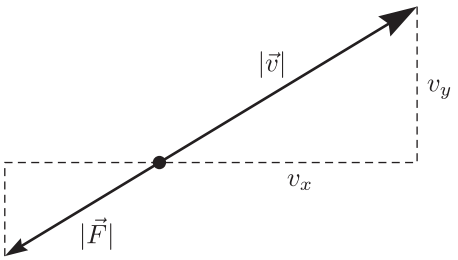



Figure 2.2: Assuming that the drag force is always opposite to the velocity vector, the similar triangles in this diagram can be used to express the force components in terms of the velocity components.

**Exercise:** Finish labeling Figure 2.2, and use it to derive formulas for the $x$ and $y$ components of the drag force, written in terms of $v_x$ and $v_y$. The magnitude of the drag force is again given by equation 2.5. (Hint: This is not an easy exercise if you've never done this sort of thing before. *Do not simply guess the answers!* You should find that the correct formula for $F_x$ involves *both* $v_x$ and $v_y$. Have your instructor check your answers before you go on.)

**Exercise:** In the space below, write the Python code for calculating the acceleration components, `ax` and `ay`, for a projectile moving in two dimensions with air resistance. The Python function for taking a square root is `sqrt()`. To square a quantity you can either just multiply it by itself, or use the Python exponentiation operator, `**`.

**Exercise:** Now think about implementing the Euler-Richardson algorithm in two dimensions. For each line in the code on page 16 that calculates a $y$ variable, you'll need to add a line to calculate the corresponding $x$ variable. But the order of these lines matters! Should you alternate $x, y, x, y \ldots$, or should you calculate all the $x$ variables first and then all the $y$ variables, or vice-versa? Explain carefully.

**Exercise:** Create a new VPython program called `Projectile2` to simulate projectile motion in two dimensions, for the specific case of a ball launched from the origin at a given initial speed and angle that are set near the top of the program. Again use a sphere to represent the ball, leaving a trail as it moves, and use a very shallow box to represent the ground. Allow for the ball to travel as far as about 150 meters in the $x$ direction before it lands, sizing the box and setting `scene.center` appropriately. Set the background to a light color for eventual printing. Use the Euler-Richardson algorithm, with a time step of 0.01 s. Run your program and check that everything seems to be working.

**Exercise:** Add code to your program to calculate and display the landing time, the value of $x$ at this time (that is, the *range* of the projectile), and the projectile's maximum height. Use interpolation for the first two quantities, as you did in `Projectile1`. For the maximum height, you'll need to test during each loop iteration whether the current height is more than the previous maximum. To do this you can use an `if` statement, whose syntax is similar to that of a `while` loop:

```
if y > ymax:
    ymax = y
```

Use `print` functions to display all three of your calculated results, along with the initial speed and angle, and the `drag` constant.

**Exercise:** Check that your `Projectile2` program gives the expected results when there is no air resistance, for a launch angle of 45° and your choice of launch speed. Record your results below, along with your calculations of the expected results.

**Exercise:** For the rest of this project there is no need to display the numerical results to so many decimal places. To round these quantities appropriately, you can use the following (admittedly arcane) syntax:

```
"Maximum height = {:.2f}".format(ymax)
```

Here we're creating a string object (in quotes) and then calling its associated `format` function with the parameter `ymax`. This function replaces the curly braces, and what's between them, with the value of `ymax`, rounded to two decimal places. (To change this to three decimal places, you would just change `.2f` to `.3f`; the `f` stands for *floating-point* format.) Make the needed changes to display all three of the calculated results to just two decimal places.

## A graphical user interface

Are you tired of having to edit your code and rerun your programs every time you want to change one of the constants or initial conditions? A more convenient approach—at least if you'll be running a simulation more than a handful of times—is to create a *graphical user interface*, or *GUI*, that lets you adjust these numbers and repeat the simulation while the program is running.

Creating a graphical user interface for a computer program can be a lot of work. You need to plan out exactly how broad a set of options to offer the user who runs your program, then design a set of graphical *controls* (buttons, sliders, etc.) to control those options, and finally write the code to display the controls and accept input from them. Fortunately, VPython makes these tasks about as easy as possible. The documentation refers to its GUI controls as *widgets*, and in this project we'll use three of them: the button, the slider, and the dynamic text widget.

Here's some minimal code to create a button that merely displays a message:

```
def launch():
    print("Launching the projectile!")

button(text="Launch!", bind=launch)
```

The first two lines define a *new function* called `launch`, and the last line creates a button that is *bound* to this function, so the function is called whenever the button is pressed.

**Exercise:** Put this code into your `Projectile2` program and try it.

You might wonder how to control *where* the button appears. VPython doesn't give you nearly as much control over the placement as would an environment for developing commercial software. By default, new widgets are placed immediately below the `scene` canvas, in what's called its *caption*. There are a few other placement options that you can read about on the Widgets documentation page if you like.

More importantly, your Launch! button doesn't yet do what we want, namely launch the projectile! In a local installation of VPython you could make it do so by moving all your initialization and simulation code into the definition of the new `launch` function. But the GlowScript environment doesn't allow a `rate` function to appear inside a bound function, so we have to do it a different way. Although it's more difficult in the present context, the following program structure will also be more useful in future projects.

The basic idea is to turn your `while` loop into an *infinite* loop that runs forever, but to execute most of the code inside the loop only if the simulation is supposed to be "running". The code on the following page provides a basic outline.

```
while True:
    rate(100)
    if running:
        # carry out an Euler-Richardson step
        if y < 0:
            running = False
            # print out results
```

The big new idea here is the introduction of a *boolean* variable (named after logician George Boole) that I've chosen to call `running`, whose value is always one of the two boolean constants `True` or `False` (note that these are capitalized in Python). When `running` is `False`, we merely call the `rate` function to delay a bit before the next loop iteration. When `running` is `True`, we carry out a time-integration step (using the code you've already written, omitted here for brevity), then test whether the ball has dropped below ground level, in which case we set `running = False` and print out the results.

**Exercise:** Insert this new code into your `Projectile2` program, replacing the two comments with the code you've already written to carry out the time step and print the results. Also add a line near the top of the program to initially set `running = True`. Test the program to verify that it works exactly as before.

**Exercise:** You're now ready to activate your Launch! button. To do so, insert the following two lines into the indented body of the `launch` function definition:

```
global running
running = True
```

Also change your initialization line near the top of the program to set `running = False` instead of `True`. Test your program again and verify that the projectile is not launched until you press the button.

**Exercise:** To allow multiple launches, simply move all of the relevant code to initialize the variables into your `launch` function. You'll also need to add their names, separated by commas, to the `global` statement. Leave the initializations of `dt` and `g` outside the function definition, since those values never change. Check that you can now launch the projectile repeatedly.

Before going on, let me pause to explain that `global` statement. In Python, any variable that you change inside a function definition is *local* by default, meaning that it exists only within the function definition and not outside it. In longer programs this behavior is a very good thing, because it frees you, when you're writing a function definition, from having to worry about which variable names have already been used elsewhere in the program. But this means that when you *do* want to change a *global* variable (that is, one that belongs to the larger program), you need to "declare" it as `global` inside your function. (JavaScript, by contrast, has the opposite behavior, making all variables global by default and forcing you to declare

them with a `var` statement if you want them to be local to a particular function.) Notice that you need to use `global` only if your function is *changing* the variable; you don't need it just to "read" the value of a variable. And for technical reasons, this means that you don't need to declare graphics objects as global variables just to change their attributes.

Now let's add some controls to let you vary the launch conditions. The best way to adjust a numerical parameter is usually with a slider control. Here is how you can add one to adjust the launch angle:

```
def adjustAngle():
    pass    # function does nothing for now
scene.append_to_caption("\n\n")
angleSlider = slider(left=10, min=0, max=90, step=1, value=45,
                     bind=adjustAngle)
```

The `slider` function creates the slider, and most of its parameters indicate which numerical values to associate with the allowed slider positions. The `value` parameter is set to an initial value (here intended to be 45 degrees), but will change when you actually adjust the slider. The `left` parameter and the `append_to_caption` function on the previous line merely insert some space around the slider in the window layout ("`\n`" is the code for "new line"). The `bind` parameter, as before, binds a function to this control; that function (`adjustAngle`) will be called whenever you adjust the slider. For now I've used Python's `pass` statement to make the function do nothing.

**Exercise:** Put this code into your program, and check that the slider appears underneath the Launch button. Then modify the code that sets the ball's initial launch velocity so it uses `angleSlider.value` instead of whatever angle you were giving it before. You should now be able to launch multiple projectiles at varying angles. Try it!

**Exercise:** The only problem now is that you can't tell exactly what angle the slider is set to—at least not until you actually click Launch! and see the output of your `print` function. To give the slider a numerical display, add the following code right after the line that creates the slider:

```
scene.append_to_caption("    Angle = ")
angleSliderReadout = wtext(text="45 degrees")
```

Then replace the `pass` statement in the `adjustAngle` function with:

```
angleSliderReadout.text = angleSlider.value + " degrees"
```

(This line uses the `+` operator to join two strings together: an example of what's called *operator overloading*.) Verify that the slider's numerical readout now works as it should.

**Exercise:** Add two more sliders, with numerical readouts, to control the ball's initial speed and the `drag` constant. Let the launch speed range from 0 to 50 m/s in increments of 1 m/s, and let the `drag` constant range from 0 to 1.0 in increments of 0.005 (in SI units). Test these sliders to make sure they work as they should.

**Exercise:** Add a second button (on the same caption line as the Launch! button) to clear away the trails from previous launches and start over. To do this, the bound function should set the ball's position back to the origin and then call the ball's `clear_trail` function (with no parameters). This button finishes your `Projectile2` program, so look over everything and make any final tweaks to the graphics, the GUI elements, the code, and the comments before turning it in.

**Exercise:** Run your `Projectile2` program, experimenting with various values of the drag coefficient, launch speed, and launch angle. You may recall from introductory physics that when there is no air resistance, the maximum range for a given launch speed occurs at an angle of 45°. Check this for a launch speed of 25 m/s, then increase the drag coefficient to 0.1 and find the angle that gives the maximum range. Record your data below, and explain why you would expect the optimum angle to be less when there is air resistance. Also make a printout of your program window, showing the trails and the numerical results from three (or more) launches with significantly different settings.

**Exercise:** The maximum speed of a batted baseball is about 110 mph, or about 50 m/s. At this speed, the ball's drag coefficient (as defined in your program) is approximately 0.005 $m^{-1}$. Using your program with these inputs, estimate the maximum range of a batted baseball, and the angle at which the maximum range is attained. Write down and justify your results below. Is your answer reasonable, compared to the typical distance of an outfield fence from home plate (about 350–400 feet)? For the same initial speed and angle, how far would the baseball go without air resistance? (Continue your answer on the following page.)

**Question:** Out of all the coding tasks and exercises you did in this project, which was the most difficult and why?

**Question:** Briefly discuss how you and your lab partner worked together on this project. Did one or the other of you find it easier to do the coding, or to understand the physics? Did you find enough time to do all your work together, or did you do some work separately? Please include an estimate of your own percentage contribution to this project (ideally 50%, but unlikely to be exactly 50%; please be as honest about this as you can).

Congratulations—you're now finished with this project! Please turn in these instruction pages, with answers written in the spaces provided, as your lab report. Be sure to attach your two printouts of the `Projectile1` and `Projectile2` program results. Turn in your code as before, either sending it by email or putting it into a public GlowScript folder and writing the folder URL below.

Physics 2300 Name _____

Lab partner _____

# Project 3: Pendulum

In this project you will explore the behavior of a pendulum. There is no better example of a system that seems simple at first but turns out to hold intricate layers of complexity.

Figure 3.1 shows the basic setup: a fixed pivot, a massless rod of length $L$, and a point-mass $m$ at the end, swinging in the plane of the page. It's easiest to analyze the motion in terms of torque and angular acceleration; recall that the angular version of Newton's second law is

$$\sum \tau = I\alpha. \tag{3.1}$$

On the left-hand side of this equation is the sum of all the torques acting on the object. On the right-hand side, $I$ is the object's rotational inertia, simply $mL^2$ for our pendulum. The angular acceleration $\alpha$ is defined analogously to the ordinary acceleration:

$$\alpha = \frac{d\omega}{dt} = \frac{d^2\theta}{dt^2}, \tag{3.2}$$

where $\omega$ is the angular velocity and $\theta$ is assumed to be in radians.

From the diagram we see that the torque due to gravity is

$$\tau_g = -L|\vec{F}_g| \sin\theta = -Lmg \sin\theta, \tag{3.3}$$

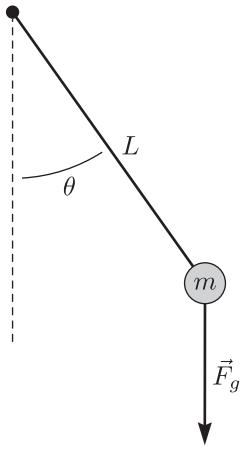



Figure 3.1: A simple pendulum of mass $m$ and length $L$, acted upon by a gravitational force $\vec{F}_g$.

where the minus sign indicates that the torque is negative (clockwise) when $\theta$ is positive, and vice-versa. If there is no friction or other torque acting, then Newton's law (equation 3.1) says simply

$$-Lmg\sin\theta = mL^2\alpha, \qquad \text{or} \qquad \alpha = -\frac{g}{L}\sin\theta. \tag{3.4}$$

Although I drew the diagram for a small positive value of $\theta$, this equation is valid for all angles—even angles greater than 90°, if the rod is rigid.

Because the mass $m$ has canceled out of equation 3.4, there will be no need to specify a mass in your pendulum simulation. You might think you *do* need to specify a length $L$, and also a value of $g$ (depending on which planet the pendulum lives on), but in fact you don't, because we still have the freedom to choose our *units* for measuring distance and time. (Yes, there are Unit Police who tell us we must always use SI units for all scientific work, but actual working scientists ignore the Unit Police and use whatever units are most appropriate for the work they're doing.) Our freedom to choose units means that whatever the actual values of $L$ and $g$, we can simply specify that we're using units in which both of these constants are equal to 1. These units are called *natural units*, because they are natural to the physics that we're studying. One advantage of using natural units is that equation 3.4 becomes simply $\alpha = -\sin\theta$. But the real advantage is that your simulation will be applicable to *any* pendulum on *any* planet. The price is that in order to apply your results to a particular pendulum, you may need to convert them from natural units to more conventional units after running the simulation.

**Exercise:** Suppose (just for this exercise) that we want to study a pendulum on earth ($g = 9.8$ m/s$^2$) that is half a meter long. Then our unit of distance will be a half meter; let's call this unit the *ham*. Define a corresponding natural unit of time, called the *tic*, such that $g = 1.0$ ham/tic$^2$. How many seconds are in a tic?

**Exercise:** In order to draw a pendulum in the VPython graphics environment, you will need to express the position of the pendulum bob in rectangular coordinates. Taking the origin to be at the pivot and the $x$ and $y$ directions to be to the right and upward, respectively, what are the formulas for $x$ and $y$ in terms of $\theta$?

**Exercise:** Write a VPython program called `Pendulum1` to simulate the motion of a pendulum swinging under the influence of gravity and no other torques. Use the Euler algorithm, with the variables `theta` (in radians), `omega`, and `alpha` playing the roles of $y$, $v_y$, and $a_y$ in your original `Projectile1` simulation. Check carefully that the order of the lines in the algorithm is the same as in the example on page 2 of Project 2. In the 3D graphics space, represent the pendulum bob as a sphere, the pendulum rod as a cylinder, and the pivot at the other end of the rod as a short cylinder perpendicular to the plane of motion. You may optionally wish to draw a stand of some sort to "support" the pivot. Use natural units with $g = L = 1$ (so you don't need these variables at all!), and a time step of 0.01 in these units. Choose initial conditions such that the amplitude of the swing will be fairly small. Once everything seems to be working, let the simulation run for a while and describe what you observe.

**Exercise:** Add a graph of $\theta$ (vertically) vs. $t$ (horizontally) to your program. Be sure to label the axes appropriately. Set the plotting `interval` to 10, in order to reduce the slow-down that will occur as more and more points are added to the graph. (Throughout the rest of this project, use your discretion to adjust the plotting interval as appropriate.) Use your graph to determine the approximate period of the pendulum in natural units, and write down the result. Does your result agree with what you learned in introductory physics? Explain carefully.

## Accuracy over long time periods

**Exercise:** Set your `Pendulum1` program to run for about 50 units of time, then run it. Sketch the appearance of the graph of $\theta$ vs. $t$ in the space below, and explain what's wrong with it.

**Exercise:** Work out a formula for the total energy (kinetic plus gravitational) of the pendulum, in terms of $\theta$, $\omega$, and constants. Take the gravitational energy to be zero at the lowest point of the pendulum's swing. (Use a picture and some trigonometry to relate $\theta$ to the height of the pendulum bob. Do not set $g = L = 1$ in this exercise.)

**Exercise:** Add some lines to your program to compute and print the total energy of the pendulum at both the beginning and the end of the simulation. For the purpose of this computation you should continue to take $L$, $g$, and $m$ all to equal 1 (in natural units). Write the results below, and explain why something must be wrong.

**Exercise:** Run the simulation again with a smaller value of `dt`. (Be sure to adjust your `rate` function and graphing `interval` accordingly.) How do the results change? Is this what you would expect?

The preceding exercises illustrate a common problem with numerical calculations: small truncation errors can add up over time to produce large inaccuracies. When a quantity such as the total energy is supposed to be exactly conserved, you should always monitor this quantity for any significant drift that might come from compounded truncation errors. While using a smaller time step can provide a

brute-force solution to this problem, it's usually better to switch to a more accurate algorithm (if possible).

**Exercise:** Modify your `Pendulum1` program to use the Euler-Richardson algorithm instead of the Euler algorithm. For a time step of 0.01 and a total running time of 50 time units, how much does the total energy drift?

At this point, please make a copy of your `Pendulum1` program and call it `Pendulum2`. The remaining changes that you'll make to `Pendulum1` will not be needed for your subsequent work, so this will save you from having to undo those changes later.

## Large-angle motion

Now that you've minimized your program's numerical inaccuracies, let's examine how the motion varies when the amplitude is changed.

Although you can read the approximate period of the pendulum from your graph, it's much more accurate to have the program calculate and print the period. Here's one way to do this: Introduce a variable called `lastTheta`, and set it equal to `theta` just before each time `theta` is updated. After updating `theta`, check whether `theta` is positive and `lastTheta` is negative. If so, the pendulum has just crossed $\theta = 0$ while moving from left to right. In this case, do an interpolation to compute the time when the crossing occurred (like when you computed the time the projectile landed in the previous project) and remember this time in another variable. The time between one one such crossing and the next equals the period of the pendulum.

**Exercise:** In your `Pendulum1` program, comment-out the code to calculate and display the pendulum's initial and final energy (by placing a `#` character in front of each line). Then implement the algorithm just described for determining the period of the pendulum. Check to make sure it works. Also modify your simulation loop so it stops right after calculating and printing the period.

**Exercise:** Modify your `Pendulum1` program to run the simulation successively for amplitudes 10°, 20°, and so on up to 170°, printing out a table with the amplitude (in degrees) in the first column and the period in the second column. To do this, you'll want to embed your entire simulation loop inside a larger `while` loop that loops over the amplitude values, reinitializing the pendulum each time through. Be sure to adjust the indentation appropriately, and be careful converting angles from degrees to radians and then back again. If you would like to reinitialize the graph of $\theta$ vs. $t$ for each new amplitude, you can do this by calling the `delete()` function

of your `gdots` object. Another useful trick is to put a tab character, `"\t"`, between the amplitude and period in your `print` output. Check that everything works, so you get a table as intended in the print area.

Of course a graph of period vs. amplitude would be better than a mere table of numbers. You could easily produce such a graph right in the GlowScript window, by adding just a few lines to your code. Often, however, for graphing final results we use a different software environment than what we used for the simulation itself. In this case, it is convenient to use an ordinary spreadsheet program.

**Exercise:** Copy your table of amplitudes and periods into a spreadsheet (Excel, Google Sheets, etc.), then make a scatter plot of period (vertically) vs. amplitude (horizontally), and label it appropriately. Arrange the graph so it fits on a single printed page with your data table, then print this page and attach it to your lab report.

Your `Pendulum1` program is now finished, so please make sure the code is well organized and adequately commented.

## Friction and driving forces

Now let's add some further complications: Friction to slow the pendulum down, and a periodic external torque that continually adds energy to the system.

A simple way to add friction is to subtract a term proportional to $\omega$ from the right-hand side of equation 3.4 for the angular acceleration:

$$\alpha = -\frac{g}{L}\sin\theta - c\,\omega. \tag{3.5}$$

A linear resistive force (or torque) of this form is called *damping*, and the coefficient $c$ is called the damping constant.

**Exercise:** In your `Pendulum2` program, remove the code that calculates and displays the initial and final energies (since we no longer expect energy to be conserved). Then add a damping term to the lines that compute `alpha` and `alphaMid`. Run the program for various values of the damping constant, ranging up to about 2.0. Describe the results briefly.

Damping removes energy from the pendulum, so the motion dies out. But we can keep the motion going by continually applying an external torque to the pendulum.

A simple yet interesting way to do this is to add a term to equation 3.5 that is sinusoidal in time:

$$\alpha = -\frac{g}{L}\sin\theta - c\,\omega + A\sin(ft). \tag{3.6}$$

This additional term is called a *driving* term. Physically, it would represent a smooth, back-and-forth twisting force, presumably applied at the pivot, that is unaffected by the pendulum's position and speed. The constants $A$ and $f$ represent the amplitude and angular frequency of the driving torque.

Because the Euler-Richardson algorithm requires two calculations of $\alpha$ (one at the beginning of the time interval and one at the middle), and because the formula for $\alpha$ is getting rather complex, it's a good idea to move this calculation into a separate function. Let's call the function `torque`, which is the same as angular acceleration in our system of units. This function should accept three parameters—$\theta$, $\omega$, and $t$—and return the value of the torque. Assuming that the constants `damp`, `driveAmp`, and `driveFreq` have already been defined, the `torque` function can be defined as follows:

```
def torque(theta, omega, t):
    return -sin(theta) - damp*omega + driveAmp*sin(driveFreq*t)
```

I should emphasize that the variables `theta`, `omega`, and `t` in this function definition (called *formal parameters*) are logically independent of the variables of the same names in the rest of your program; here we could just as well call them Larry, Moe, and Curly (at least as far as the computer is concerned). When you call the `torque` function from elsewhere in your program you can pass it any values you like for these three variables, and it will return the appropriate torque.

**Exercise:** Add this function definition to your program, and use it appropriately each time your program computes the angular acceleration. (Be sure to pass the correct values to your `torque` function for the mid-point calculation!) Test your program with the damping constant equal to 0.5, the drive amplitude equal to 0.5, and the drive frequency equal to 2/3. You should find that after an initial "transient" behavior, the motion becomes periodic. What is the approximate period of the motion, and what is the significance of this value?

## Chaos!

**Exercise:** Now increase the drive amplitude to 1.2, and run the program again (for about 100 time units). Describe the behavior briefly.

When the drive amplitude is sufficiently large, the pendulum can swing over the top of the pivot and the motion becomes much more complex. In some cases, such as the one you just saw, the motion never settles down to become periodic. Because the motion seems so unpredictable, the word "chaos" often comes to mind.

*Chaos* is actually a technical term in dynamics, used to describe behavior that is unpredictable in practice because even a tiny change in the initial condition results in an exponentially increasing change in the motion. A good way to test for chaotic behavior is to run a simulation twice, with two slightly different initial conditions, and monitor the difference in the motion as a function of time.

**Exercise:** Make a copy of your `Pendulum2` program and call it `Pendulum3`, for later use. Then modify `Pendulum2` to simulate the motion of *two* damped, driven pendulums simultaneously. Use two sets of variables for the two separate pendulums, and simulate both motions using a single `while` loop in your code. Call the new variables `theta2`, `omega2`, and so on. I'll refer to the angle of the original pendulum as $\theta_1$, but you may leave it as simply `theta` in your code. In the 3D graphics space, put the second pendulum in front of the first, suspended from the same pivot. On your graph, plot both $\theta_1$ and $\theta_2$ vs. time, using two different colors (it's a nice touch to use the same colors for the pendulums in the 3D graphics space that you use on the graph). Start one pendulum at $\theta = 0$ and the other at $\theta = 0.001$ (radians), with $\omega = 0$ initially for both, and run for about 200 units of time. Use the same damping and driving constants as in the two previous exercises, and describe the results for both values of the driving amplitude. Print the more interesting of the two graphs, and attach it to your lab report. (To print a graph, first make a screen capture of just the graph, then open it in a suitable graphics program and print it from there. Consult with your instructor if you are unsure of how to do this.)

**Exercise:** To better see how the motions of the two pendulums relate to each other, modify your `Pendulum2` program to produce a second graph, plotting $\ln|\theta_2 - \theta_1|$ vs. $t$. The VPython function for the natural logarithm is `log()`. Run your program using the same constants as before, including both values (0.5 and 1.2) of the drive amplitude. This time it should suffice to run for about 80 units of time. Print the log-difference plot for both cases, and discuss the results in some detail. Why do both plots have several downward-pointing spikes?

The roughly exponential behavior of $\theta_2 - \theta_1$ as a function of time is a feature of many dynamical systems. We can write this behavior as

$$\theta_2 - \theta_1 \approx (\text{constant}) \times e^{\lambda t}, \tag{3.7}$$

where $\lambda$ is called a *Lyapunov exponent*. When $\lambda$ is positive the system is chaotic: even the smallest difference in initial conditions will grow over time until the two systems have radically different behavior. Since we can never know the initial conditions of a real physical system with infinite accuracy, it is effectively impossible to predict the behavior of a chaotic system over long time periods. Although our damped and driven pendulum is a somewhat contrived system, chaotic behavior is also quite common in the real world.

**Exercise:** Estimate the Lyapunov exponent of the pendulum system for each of the values of the constants used in the previous exercise. (Hint: Use a ruler to draw a reasonable straight line to fit your logarithmic graphs of $\theta_2 - \theta_1$, ignoring the downward-pointing spikes.)

Your `Pendulum2` program is now finished, so check everything over and be sure that the code is well organized and adequately commented.

### When is the motion chaotic?

So far you have explored the motion of the damped and driven pendulum for only two sets of damping and driving parameters—one that produces chaotic behavior and one that doesn't. Your final task will be to map out in more detail which conditions produce chaos.

In principle you could vary the damping constant, the driving frequency, *and* the driving amplitude—but thoroughly exploring that three-parameter space would take a long time. Instead, you will keep the damping constant and the driving frequency fixed at the same values as before, and vary only the driving amplitude.

**Exercise:** In your `Pendulum3` program (which simulates only a single damped and driven pendulum), change the simulation loop to run indefinitely, but add a GUI button to pause and resume the simulation. Then add a slider to adjust the driving amplitude to any value between 0 and 2.0, in increments of 0.01. Be sure to create a numerical readout for the slider. Test these controls to make sure they work.

**Exercise:** Change the graph in `Pendulum3` to plot $\omega$ vertically vs. $\theta$ horizontally, instead of $\theta$ vs. $t$. This new graph is called a *phase space plot*. Use the `xmin` and `xmax` parameters of the `graph` function to set the horizontal scale on the graph to run from $-\pi$ to $+\pi$, and then, in your simulation loop, insert some `if` statements to shift $\theta$ by $2\pi$ whenever necessary to keep it within the range of the graph. (This doesn't affect the physics, because changing an angle by $2\pi$ doesn't really change it.) Set the `size` parameter of the `gdots` object to 1, and adjust the plotting interval to compromise between a more complete plot and a reasonable execution speed. Finally, add another GUI button to the program that clears all dots from the plot, by calling the `delete` function of your `gdots` object. Again, test everything to make sure it works.

**Exercise:** Run your simulation and systematically explore what happens as the drive amplitude increases from 0 to 2. For some settings the motion will settle into a repeating pattern (once any "transient" behavior has died out), while for others it will be chaotic, never exactly repeating itself. Print a few of the more interesting phase space plots (combining them onto a single page if you know a way to do that), being sure to label them with the drive amplitude settings. Describe the motion in words for each of these cases. Also, in the space below, make a list of the drive amplitude values for which the motion is chaotic.

This exercise completes Project 3. Be sure to attach all the printouts to this lab report, and to submit your code by email (or by emailing a link).

Physics 2300 Name _____

Lab partner _____

# Project 4: Orbits

In this project you will simulate the motion of planets orbiting the sun.

The only important force in this situation is gravity. According to Newton, the gravitational force between two objects is always attractive, with magnitude

$$|\vec{F}_g| = \frac{Gm_1m_2}{r_{12}^2}, \tag{4.1}$$

where $m_1$ and $m_2$ are the masses of the two objects and $r_{12}$ is the distance between them. The associated potential energy, for later reference, is

$$U_g = -\frac{Gm_1m_2}{r_{12}}. \tag{4.2}$$

This formula takes some getting used to, because it is always negative. The important thing, though, is that as two objects get farther apart their potential energy increases (becomes less negative).

We'll start with a single planet orbiting the sun. Because the sun is so much more massive than any planet, it's a reasonable approximation to neglect the sun's motion. Then we can put the sun at the origin of our coordinate system, and the formula for its force on the planet will simplify.

**Exercise:** Use equation 4.1 and an appropriate diagram to find the $x$ and $y$ components of the sun's gravitational force on a planet, in terms of the planet's coordinates $x$ and $y$. *Be sure that you have the correct formulas before you try to write any code!*

## Units

Rather than using SI units, it's usually a good idea to choose units that are natural for the problem being solved. In the case of planets orbiting the sun, a natural unit of distance is the *astronomical unit* (AU), defined as the average distance between the earth and the sun (about 150 million kilometers). A natural unit of time is the year ($3 \times 10^7$ seconds), and a natural unit of mass is the sun's mass ($2 \times 10^{30}$ kg).

**Exercise:** What is earth's orbital speed in these natural units (assuming its orbit to be circular)? (In this exercise and the next, do not start with SI units and then convert; there's a much easier method.)

**Exercise:** Find the value of $G$ in these natural units, by applying Newton's laws to earth's orbit. (You'll need to recall or look up the formula for the acceleration of an object moving in a circle at constant speed.)

**Exercise:** Suppose that a planet is in a circular orbit of radius $r$. Find a formula for its speed in terms of $r$, using natural units to simplify your answer.

## A one-planet simulation

**Exercise:** Write a VPython simulation program called `Orbit1`, for a single planet orbiting the sun in the $xy$ plane. Represent the planet as a sphere in the 3D simulation space, with a larger sphere at the origin to represent the sun. (Choose the radii of these spheres for easy visibility—not based on their sizes in the actual solar system!) Have the planet leave a trail to mark its orbital path. Also create two thin cylinders to represent the $x$ and $y$ axes, making each of them a few AU long. In this project you will always need to view the orbits directly from the perpendicular ($z$) direction, so set `scene.userspin = False` to disable rotating the view, and set `scene.fov = 0.01` to give a very small field of view (0.01 radians), as if you are viewing the scene from very far away through a telescope. Set the background color to white, since you'll be printing some of your orbital images. To simulate the planet's motion use the Euler algorithm for now, with a time step of 0.01 (in years). Start the planet at $x = 1.5$, $y = 0$, and an initial `vx` and `vy` that should give a circular orbit. Describe the result of your simulation for a running time of ten years. Try the simulation again with a time step of 0.001 (adjusting the `rate` parameter and the trail's `interval` accordingly), and discuss the implications of your results.

**Exercise:** Add a `totalE` function to your program, which computes and returns the planet's total energy per unit mass. Use equation 4.2 for the potential energy. Your function doesn't need to have any parameters, because $x$, $y$, $v_x$, and $v_y$ are all global variables. Add code both before and after your simulation loop to call the `totalE` function and print out the result, labeled with appropriate text. Write down the results for a couple of different values of `dt`, and comment briefly.

**Exercise:** By now you should be anxious to replace the Euler algorithm in your simulation with something more accurate. Try the Euler-Richardson algorithm next (but comment-out your Euler algorithm code, rather than deleting it, so it will still be visible), and write down the results (initial and final energy) for a couple of values of `dt`. Comment briefly. (You should find that the Euler-Richardson algorithm is much more accurate than the Euler algorithm.)

## The Verlet algorithm

The Euler-Richardson algorithm is *so* much better than the Euler algorithm that you may be tempted to settle for it. However, in problems such as this where the force depends only on the positions of the particles (not on their velocities), there is another algorithm that usually does significantly better still, and is no harder to code.

You may have already noticed, while coding the Euler-Richardson algorithm, that there's no actual need to know the velocity at the interval's midpoint when this velocity won't be needed to calculate the force. In this case, you could instead combine this velocity calculation with the calculation of the updated position. So, for the $x$ components, the equations

$$v_{x,\text{mid}} = v_{x,\text{initial}} + \tfrac{1}{2}a_{x,\text{initial}}dt \qquad \text{and} \qquad x_{\text{final}} = x_{\text{initial}} + v_{x,\text{mid}}dt \tag{4.3}$$

can simply be combined into the single formula

$$x_{\text{final}} = x_{\text{initial}} + v_{x,\text{initial}}dt + \tfrac{1}{2}a_{x,\text{initial}}(dt)^2. \tag{4.4}$$

You should recognize this formula from introductory physics: It is the *exact* formula for the motion of an object whose acceleration is *constant*. We'll continue to use it (for small $dt$) even when the acceleration isn't constant, as we've already been doing, in effect, with the Euler-Richardson algorithm.

Our improvement on the Euler-Richardson algorithm will be in the calculation of the updated velocity. Once we've updated the position using equation 4.4, we can use this updated position to calculate the acceleration at the *end* of the time interval (so long as the force depends only on the position, not on the velocity). We can then estimate the *average* acceleration as the average of the initial and final accelerations, and use this average to update the velocity:

$$v_{x,\text{final}} = v_{x,\text{initial}} + a_{x,\text{average}}dt \approx v_{x,\text{initial}} + \Big(\frac{a_{x,\text{initial}} + a_{x,\text{final}}}{2}\Big)dt. \tag{4.5}$$

This estimate of the average acceleration is more symmetrical, and hence more accurate in most situations, than the Euler-Richardson method of calculating $a_x$ from an estimated value of $x_{\text{mid}}$. The combination of equations 4.4 and 4.5 is known as the *Verlet algorithm*, or alternatively as the *second Taylor approximation* (STA). For two-dimensional motion, of course, each step would entail updating the $y$ components as well as the $x$ components.

The slick way to implement the Verlet algorithm is to break equation 4.5 into two parts, separated by the calculation of the new acceleration, as follows:

0. Before the loop begins, calculate the initial acceleration.
1. Update the position using equation 4.4.
2. Update the velocity half-way, adding $\frac{1}{2}a_x dt$.
3. Update the acceleration, calculating it from the new position.
4. Finish updating the velocity, again adding $\frac{1}{2}a_x dt$.
5. Repeat steps 1 through 4 in each loop iteration.

Notice that this procedure requires only *one* evaluation of the acceleration for each time interval. In the next project, where execution speed will be an issue, this will be another significant advantage over the Euler-Richardson algorithm.

**Exercise:** Comment-out the Euler-Richardson code in your `Orbit1` program, then implement the Verlet algorithm. Since the acceleration needs to be calculated once outside the loop and once inside it, move this code into a separate function and call it from both places. (The function needn't have any parameters, or return any value; just make `ax` and `ay` global variables.) Test your program for the same initial conditions, running time, and values of `dt` that you used in the previous exercise, and compare the accuracy to which the two algorithms conserve energy, writing your results below. Be sure to display enough decimal places in the energy values to allow you to see the change!

### Kepler's laws

In the early 1600s, Johannes Kepler was the first to discover simple mathematical laws to accurately describe the observed motions of the planets. Nowadays we number Kepler's important discoveries 1, 2, and 3; I like to throw in a 0th law that may have gone without saying in Kepler's time, but isn't at all obvious from a modern perspective:

0. Planetary orbits are closed paths; each planet returns to the same point after one full orbit.

1. The shape of each orbit is an ellipse, with the sun at one focus of the ellipse.

2. The planets move faster when they are closer to the sun, in such a way that a line drawn from the sun to any planet sweeps out equal areas in equal times.

3. The outer planets move slower than the inner ones, in such a way that the cube of the length of the ellipse's semimajor axis is proportional to the square of the period of the orbit.

Later, in 1687, Isaac Newton published a book showing how all of Kepler's laws (and much more) can be *deduced* from his more fundamental laws of motion and gravity. Even today, however, a rigorous deduction of Kepler's laws from Newton's laws requires some sophisticated and lengthy calculations. On the other hand, you now have a computer program that simulates planetary motion according to Newton's laws. Let's check, then, whether the resulting motion obeys Kepler's laws.

**Exercise:** (Kepler's 0th Law.) If you haven't already, try out some other initial conditions for your simulated planet. A good way to start is to make the initial values of $x$, $y$, and $v_x$ the same as for earth, and vary the value of $v_y$ from slightly lower than earth's speed to slightly higher. Describe the results in the space below. Does your simulated planet obey Kepler's zeroth law? How small a value of `dt` must you use to obtain consistent results?

**Exercise:** (Kepler's 1st Law.) An ellipse is defined as the set of all points for which the sum of the distances to the two foci is a constant. According to Kepler's first law, one focus of the ellipse should be at the sun, which is at the origin of your coordinate system. For the initial conditions suggested in the previous exercise, the other focus should be symmetrically located exactly 1 AU from the left end of the ellipse. (The second focus may lie either left or right of the first.) Adjust your initial conditions to obtain a reasonably elongated orbit, and zoom in or out until the orbit fills most of the scene. Print this image, making sure it fills most of a page, and label the two foci on the printout. Then pick at least three dissimilar points along the orbit and for each, use a ruler to measure the sum of the distances to the two foci. Show your measurements and calculations on the printout, and discuss (on the printout) whether Kepler's first law seems to hold for your orbit.

**Exercise:** (Kepler's 2nd Law.) Modify the `interval` setting of your moving planet so that only two or three dozen trail points appear around the orbit. Use the same `dt` and initial conditions as in the previous exercise. Run the simulation for just a single orbit and again print the resulting image, making sure it fills most of a page. Then pick three dissimilar intervals along the orbit, identical in duration, and for each, use a ruler to determine the approximate area swept out by an imaginary line from the sun to the planet. Show your measurements and calculations on the printout, and discuss (on the printout) whether Kepler's second law seems to hold for your orbit.

**Exercise:** (Kepler's 3rd Law.) The semimajor axis of an ellipse is defined as half of its widest width. As long as you use an initial $v_x$ of zero, this width should be along the $x$ axis of your image. You can then test Kepler's third law as follows. Add code to your program to determine when the planet crosses the $x$ axis, and for each crossing, to print out (to the screen) the value of $x$ and the time. From these data, you can calculate (by hand is fine) both the semimajor axis and the period of the orbit. Run the simulation for three dissimilar orbits, recording the data below. Then for each orbit, calculate the cube of the semimajor axis and the square of the period. Discuss whether your results are consistent with Kepler's third law.

### Elongated orbits and variable time steps

By now you may have noticed that the closer your simulated planet gets to the sun, the smaller you need to make `dt` to reduce truncation errors to an acceptable level. For highly elongated orbits this is awkward, because the small value of `dt` is needed only near one end of the orbit, while a much larger value of `dt` would suffice elsewhere.

Fortunately, there's no law that says we have to use the same value of `dt` throughout the simulation. Instead, we can use what is called *adaptive step size control* to continually adjust `dt` as appropriate. Nearly all of the "professional quality" algorithms for numerically solving differential equations employ some sort of adaptive step size control. Most of these algorithms are quite complex and sophisticated—not worth our time in a first course on computer simulations. In your `Orbit1` program, however, there is a very simple way to add adaptive step size control.

Think about it: We want `dt` to be small when the planet is close to the sun (where its acceleration is large and rapidly changing) but large when the planet is farther away (where its acceleration is small and slowly changing). A natural way to accomplish this would be to make `dt` proportional to $r$, or to some positive power of $r$. Equivalently, we could make `dt` proportional to some negative power of the acceleration, $|\vec{a}|$, which is proportional to $1/r^2$:

$$\texttt{dt} \propto |\vec{a}|^{-n} \propto r^{2n}. \tag{4.6}$$

Let's work with $|\vec{a}|$, since this will make it easier to generalize our approach to multi-planet simulations in the next section. The optimum power of $|\vec{a}|$ is not easy to guess, but I've done a bit of pencil-and-paper analysis that seems to indicate that $n = 1$ is a good choice. In practice, this choice seems to work just fine.

**Exercise:** Modify your `Orbit1` program to set `dt` equal to a constant (call it `tolerance`) times $|\vec{a}|^{-1}$. This should be done at the beginning of each loop iteration. To select a good value of `tolerance`, either do a quick calculation or just try some values and see what works. Explain how you chose your value of `tolerance` in the space below.

**Exercise:** Use your `Orbit1` program to model the orbit of Halley's comet, which is 0.58 AU from the sun at its closest approach and has a period of 76 years. What is its maximum orbital speed in AU/year? How far from the sun is the orbit's most distant point? What did you have to do to determine these quantities?

Your `Orbit1` program is now finished, so please make sure the code is well organized, easy to read, and adequately commented.

## A two-planet simulation

Don't let this get you down, but all the results of your `Orbit1` program were already known to Newton more than 300 years ago. The real advantage of a computer simulation over pencil-and-paper methods comes at the next stage, when we want to model a system that is more complex than a single particle subject to an inverse-square force. For instance, you could easily explore what happens when the force law is modified, either in some hypothetical universe or in the real solar system due to the sun's nonspherical shape or the effects of general relativity. An even more difficult problem, though, is to predict the motion of three or more mutually gravitating objects. With the exception of a few unrealistically symmetrical configurations, there are no analytic formulas for the motions of such systems.

Your next task will be to write a new computer program (`Orbit2`) to model the behavior of *two* planets orbiting the sun, including the effects of their mutual gravitational attraction. Because these effects are often small, you'll need to run the simulation for a relatively long time—long enough to observe dozens or hundreds of orbits.

**Exercise:** Make a copy of your `Orbit1` simulation and call it `Orbit2`. In `Orbit2`, remove the commented-out code for the Euler and Euler-Richardson algorithms, as well as the code that prints the initial and final energies (but keep the function that calculates the energy). You may also remove the cylinders that represent the $x$ and $y$ axes. In anticipation of adding a second planet, change the planet's variable names to `x1`, `y1`, `vx1`, and so on. Set the simulation loop to run indefinitely, but introduce a boolean `running` variable and a "Start/stop" button that you can use to pause and resume the simulation. Change the planet's initial position and velocity to simulate earth's orbit (for now). Spend some time adjusting the graphics settings, including the size of the planet and its trail (which you could leave as "points" or change to "curve"), and the trail interval and animation rate. You want to be able to simulate dozens of orbits quickly, and to discern small changes from one orbit to the next.

**Exercise:** Add `wtext` objects to `Orbit2` to display the elapsed time and the system's total energy in the space below the graphics scene. Update these readouts during each simulation loop iteration, and use the string `format` function to round each of them to an appropriate number of decimal places.

**Exercise:** Now add a second planet to your simulation, orbiting like the first under the influence of the sun's gravitational pull. Don't worry yet about including the gravitational force *between* the two planets. For simplicity, please assume that the second planet also orbits in the $xy$ plane, and start it out in a circular orbit of some reasonable size. In the line that calculates `dt`, use the *larger* of the two planets' accelerations. (Use the `max` function.) Draw the two planets and their trails in

different colors. Be sure to update your energy function to include the second planet's energy.

**Exercise:** Finally, modify your simulation code to include the gravitational force between the two planets. Up until now the masses of the planets have been irrelevant, but at this point you'll have to introduce variables to represent the masses (in units of the sun's mass). To calculate the force, it's helpful to first calculate the $x$ and $y$ separations between the two planets, storing these in temporary variables. Be very careful to check that you're using the correct formula for the force. Also modify your energy calculation to incorporate the planets' masses and the inter-planet potential energy. To test your program, start the planets in orbits that would be circular if they did not interact, with radii of 1 and 1.5 (about right for earth and Mars). First set both masses equal to $10^{-6}$, and verify that the orbits remain essentially circular. Then set both masses equal to 0.02 (about 20 times the mass of Jupiter), and describe what happens. Is the total energy reasonably constant?

Your `Orbit2` program is now finished! It can be used to explore a great variety of scenarios; feel free to experiment. The following exercise should give you a good start.

**Exercise:** Set the masses of your two planets to 0.001 and $10^{-8}$ (`1e-8` in Python syntax), to simulate Jupiter and an asteroid. Start both in orbits that would be circular if they did not interact with each other, with Jupiter at 5.2 AU from the sun and the asteroid at 3.0 AU. Run the program for a few hundred simulated years and describe the resulting orbits. Then repeat the simulation with the asteroid at 3.1 AU, 3.2 AU, and so on up to 4.0 AU (adjusting the asteroid's speed to keep its orbit circular if Jupiter weren't there), and describe the results. For what sizes of the asteroid's orbit does the influence of Jupiter appear to be the greatest? Can you explain why? Print out two representative images, labeling each with the asteroid's initial orbit size. Observations of the actual asteroid belt show that there are gaps in it, where very few asteroids are found. At what distances from the sun would you expect to find gaps? (Continue your answer on the following page.)

**Question:** Which part of this project did you find most difficult? Any other comments on how it went?

**Question:** How well did you and your lab partner work together on this project? What is your estimate of your own percentage contribution, and your lab partner's?

Congratulations! You are finished with this project. Please turn in your code by email or by emailing a public link, as usual.

[This page intentionally left blank.]

Physics 2300 Name _____

Lab partner _____

# Project 5: Molecular Dynamics

If a computer can model three mutually interacting objects, why not model more than three? As you'll soon see, there is little additional difficulty in coding a simulation of arbitrarily many interacting particles. Furthermore, today's personal computers are fast enough to simulate the motion of hundreds of particles "while you wait," and to animate this motion at reasonable frame rates while the calculations are being performed.

The main difficulty in designing a many-particle simulation is not in the simulation itself but rather in deciding what we want to learn from it. Predicting the individual trajectories of a hundred particles is usually impractical because these trajectories are chaotic. Even if we could make such predictions, the sheer amount of data would leave us bewildered. Instead, we'll want to focus on higher-level patterns and statistical data.

In this project we'll also shift our attention from the very large to the very small—from planets to molecules. The main goal will be to learn how the familiar properties of matter arise from motions at the molecular scale.

## Molecular forces

Under ordinary circumstances, molecules are electrically neutral. This implies that the forces between them are negligible when they're far apart. However, when two molecules approach each other, the distortion of their electron clouds usually produces a weak attractive force. When they get too close, the force becomes strongly repulsive (see Figure 5.1). For all but the simplest molecules, these forces also depend on the molecules' shapes and orientations. In this project we'll ignore such complications and just simulate the behavior of spherically symmetric molecules such as noble gas atoms.

Even for the simplest molecules, there is no simple, exact formula for the intermolecular force. Fortunately, we don't really need an exact formula; any ap-

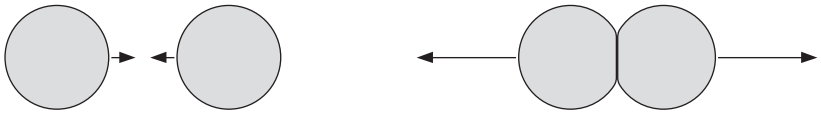


Figure 5.1: When two molecules are near each other, there is a weak attractive force between them. When they get too close, the force becomes strongly repulsive.

proximate formula with the right general behavior will give us useful results. The formula that is most commonly used to model the interaction between noble gas atoms is the *Lennard-Jones potential*,

$$U(r) = 4\epsilon\left[\left(\frac{r_0}{r}\right)^{12} - \left(\frac{r_0}{r}\right)^{6}\right]. \tag{5.1}$$

This is a formula for the potential energy, $U(r)$, in terms of the distance $r$ between the centers of the two interacting molecules. The constants $r_0$ and $\epsilon$ represent the approximate molecular diameter and the overall strength of the interaction. The numerical values of these constants will depend on the specific type of molecule; the table below gives some values obtained by fitting the Lennard-Jones model to experimental data for noble gases at low densities.

| | $r_0$ (Å) | $\epsilon$ (eV) |
|---|---|---|
| helium | 2.65 | 0.00057 |
| neon | 2.76 | 0.00315 |
| argon | 3.44 | 0.0105 |

Figure 5.2 shows a graph of the Lennard-Jones potential. When $r \gg r_0$, the energy is negative and approaches zero in proportion to $1/r^6$. This is the correct behavior of the so-called van der Waals force, and can be derived from quantum theory. When $r < r_0$ the energy becomes positive, rising very rapidly to give a strong repulsive force as $r$ decreases. John Lennard-Jones suggested using a power law to model this repulsive behavior, and nowadays we normally take the power

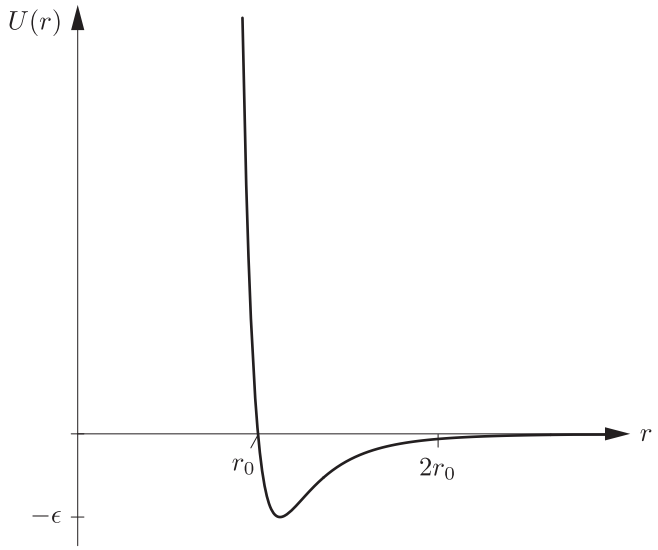



Figure 5.2: The Lennard-Jones potential energy function (equation 5.1).

to be $-12$ for computational convenience. We'll stick with this choice because no simple improvement to it would give significantly more accurate results.

**Exercise:** Find the value of $r$ (in terms of $r_0$) at which the Lennard-Jones function reaches its minimum, and show that its value at that point is $-\epsilon$.

**Exercise:** Consider two molecules, located at $(x_1, y_1)$ and $(x_2, y_2)$, interacting via the Lennard-Jones force. Find formulas for the $x$ and $y$ components of the force acting on each molecule, in terms of their coordinates and their separation distance $r$. (Hint: First calculate the $r$ component of the force, which is given by the general formula $F_r = -dU/dr$. Then draw a picture, and be careful with minus signs.)

## Units

Once again we can make our lives easier by choosing units that are natural to the system being modeled. For a collection of identical molecules interacting via the Lennard-Jones force, a natural system of units would set $r_0$, $\epsilon$, and the molecular mass ($m$) all equal to 1. These three constants then become our units of distance, energy, and mass. Units for other mechanical quantities such as time and velocity are implicitly defined in terms of these.

**Exercise:** What combination of the constants $r_0$, $\epsilon$, and $m$ has units of time?

**Exercise:** If we are modeling argon atoms using natural units, what is the duration of one natural unit of time, expressed in seconds? (Use the values of $r_0$ and $\epsilon$ from the table above. Note that the $r_0$ values are given in Ångström units (Å), where 1 Å $= 10^{-10}$ m, while the $\epsilon$ values are given in electron-volts (eV), where 1 eV $= 1.60 \times 10^{-19}$ J.)

**Exercise:** Suppose that an argon atom has a speed of 1, expressed in natural units. What is its speed in meters per second?

Another quantity that we'll eventually want to determine for our collection of molecules is temperature. To define a natural unit of temperature we can set Boltzmann's constant $k_B$ (which is essentially a conversion factor between molecular energy and temperature) equal to 1. In conventional units,

$$k_B = 1.38 \times 10^{-23} \text{ J/K} = 8.62 \times 10^{-5} \text{ eV/K}. \tag{5.2}$$

I'll explain later how to actually determine the temperature of a system from the speeds of the particles.

**Exercise:** Suppose that a collection of argon atoms has a temperature of 1 in natural units. What is its temperature in kelvin? (For comparison, the boiling point of argon at atmospheric pressure is 87 K.)

**Exercise:** Repeat the previous exercise for helium, and discuss the result briefly.

## Lists

To simulate the motion of just *two* noble gas atoms, you could simply modify your `Orbit2` program to use the Lennard-Jones force law instead of Newton's law of gravity. Recall that in that program you used the variables `x1`, `y1`, `vx1`, and so on for the first planet, and `x2`, `y2`, `vx2`, and so on for the second planet. As you can imagine, this approach becomes awkward if you add more planets (or atoms). Fortunately, Python (like nearly all programming languages) provides a convenient mechanism for working with a collection of related variables: a *list* (also sometimes called an *array*), which you refer to by a single variable name such as `x` or `vx`.

The elements of a Python list are numbered from 0 up to some maximum value. To access a particular element, you put that element's number in square brackets after the name of the list:

```
x[14] = 4.2
fib[2] = fib[0] + fib[1]
```

The number in brackets is called an *index*. Because the index of a list's first element is zero, the index of the last element is always one less than the size of the list. For example, if `x` is a list of 100 numbers, then you access those numbers by typing `x[0]`, `x[1]`, and so on up to `x[99]`. (This convention is common to Python and all the C-derived languages, including Java and JavaScript. But there are other languages, such as Fortran and Mathematica, in which list indices start at 1 rather than 0.)

This bracket-index notation would be no improvement at all if the quantity in brackets always had to be a literal number. But it doesn't: You're allowed to put any integer-valued variable, or any integer-valued expression, inside the brackets.

So, for instance, if `x` is a list of 100 elements, you could add up all the elements with this `while` loop:

```
sum = 0
i = 0
while i < 100:
    sum += x[i]
    i += 1
```

Notice that the condition on the loop is a strict `<`, not `<=`, because `x[100]` doesn't exist.

Although there's nothing wrong with the preceding `while` loop, Python (like most programming languages) provides a more compact way of executing a block of code a fixed number of times: a `for` loop. To add up the elements of the `x` list with a `for` loop, you would type the following:

```
sum = 0
for i in range(100):
    sum += x[i]
```

This code also uses Python's `range` function, which in this case effectively produces a list of the integers 0 through 99 (not including 100!). Both the `range` function and the `for` loop can be used in other ways, and are a bit difficult to explain in general, but for looping over the elements of a list, this example is all you need to know. (If you've used `for` loops in other languages, you may need to revise some of your expectations about how you can and cannot use a `for` loop in Python. I prefer to use `for` to loop over list elements, and in other situations where an integer variable is used to count some pre-determined number of iterations; for any other looping situation I use `while`.)

And how do you create a Python list in the first place? If the list is sufficiently short, you can create it by providing literal values, in brackets, separated by commas:

```
daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

Often, though, we want to initialize a list in a more automated way, and awkwardly, the best method in that case depends on what Python environment you're using. The most robust method, I think, is to first create an empty list and then write a loop to add elements to it using the list's `append` function:

```
x = []
for i in range(100):
    x.append(0)
```

In this example I've given every list element an initial value of 0, but you could pass any other initial value (possibly depending on `i`) to the `append` function.

## The program design

You're nearly ready to start writing a molecular dynamics simulation program. But there are several decisions to make before you can actually start coding, and to save time I've made some of these decisions for you.

First decision: You'll simulate a collection of noble gas atoms in *two* dimensions, not three. A 3D simulation would be only a tiny bit harder to code in GlowScript-VPython, and you can certainly try it later if you like, but in 3D it's hard to see what's happening, because the atoms in front tend to block your view of the ones behind. Fortunately, there's plenty to be learned from a 2D molecular dynamics simulation.

Second decision: The atoms will live in a square "box" whose walls are located at the edges of the graphics scene. So you should set `scene.width` and `scene.height` to be the same; you can decide exactly how many pixels they should be, depending on your screen size. The $x$ and $y$ coordinates inside the box will each run from 0 up to a maximum value that we'll call `w` (for *width*), measured in natural units as defined above (multiples of $r_0$). You'll set `w = 20` initially, but plan on changing the value later. Then you should set `scene.center` to be at the middle of the box, and set `scene.range` to put the edges at 0 and `w`. Set `scene.fov` to a small value like 0.01, to eliminate distortion as in the previous project. Disable auto-scaling, zooming, and rotation.

Third decision: Use the variable name `N` for the total number of atoms; use `x`, `y`, `vx`, `vy`, `ax`, and `ay` for the lists that will hold the atoms' positions, velocities, and accelerations; and use a list called `ball` for the `sphere` objects that will represent the atoms in the graphics scene. Set `N = 100` for now, but write your code to work for any reasonable value of `N`.

**Exercise:** Create a new GlowScript program called `MD`, and put code in it to implement the design decisions described above. Use a `for` loop to initialize all the velocities and accelerations to zero, to initialize the positions to a common location near the middle of the box, and to initialize each element of the `ball` list to a `sphere` at the corresponding position, with diameter 1.0, in your favorite color. Run the program and check that you see a single sphere at the correct position, with the correct size. (You'll see only one sphere, because they're all at the same position.)

**Exercise:** Modify your initialization code to place all the atoms at different positions inside the box, so none of the spheres that represent them overlap. The easiest way to do this is to arrange them in regular rows, placing each atom next to the previous one but starting a new row whenever the old row is full. I suggest that you place them pretty close together, rather than trying to spread them uniformly over the entire box. You'll need a couple of variables to keep track of the $x$ and $y$ locations where the next atom goes (or where the last one went). Spend some time on this task, trying out your ideas to see if they work, but if you and your lab partner can't come up with a working algorithm within 15 minutes or so, be sure to ask someone else for a hint. Once you get your code working for `N = 100` and

`w = 20`, change these values a few times and make sure it still works—but don't worry about what happens when the box is too small to comfortably hold all the atoms.

## Coding the simulation

Now you can start adding code to put the atoms into motion.

**Exercise:** Create an infinite `while` loop for your main simulation loop. Include a `rate` function with a parameter of 1000 or more, so the loop will run as fast as possible. During each iteration this loop should call a function called `singleStep` about 10 times; this is the number of calculation steps per animation frame (and you can fine-tune it later). After all 10 calls to `singleStep`, update the positions of all the `ball` objects using the current values of `x` and `y`. In the `singleStep` function itself, put in some temporary code that merely changes `x[0]` and `y[0]` by a tiny amount. Test your program to verify that the first atom-sphere moves as expected.

**Exercise:** Now remove the temporary code from `singleStep` and replace it with code to implement the Verlet algorithm: First update all the positions and update the velocities half-way, then call a separate function (call it `computeAccelerations`) to compute all the new accelerations, and finally update the velocities the remaining half-way. Use a fixed time step of 0.02 in natural units. Create the `computeAccelerations` function, but for now, just put some temporary code in it that sets `ax[0]` and `ay[0]` to some tiny nonzero values. Again, test your program to make sure it behaves as expected.

The `computeAccelerations` function must have two parts: one to calculate the accelerations from collisions with the walls of the box, and one to calculate the accelerations from collisions between atoms. The collisions with the walls are potentially trickier, especially if we want infinitely stiff walls that produce instantaneous changes in velocity. Instead, it's easier to make the walls "soft," so the force they apply increases gradually as the edge of an atom moves farther into the wall. A linear "spring" force works well; here is a code fragment that implements such a force for the vertical walls:

```
for i in range(N):
    if x[i] < 0.5:
        ax[i] = wallStiffness * (0.5 - x[i])
    elif x[i] > (w - 0.5):
        ax[i] = wallStiffness * (w - 0.5 - x[i])
    else:
        ax[i] = 0.0
```

The Python word `elif` is short for "else if"; I hope this example of its use is self-explanatory. Here `wallStiffness` is the "spring constant," for which a value of 50 in natural units works pretty well. Notice that since $m = 1$, the force on a molecule is the same as its acceleration.

**Question:** Why does the `if` statement in this code test whether `x[i]` is less than 0.5, rather than testing whether `x[i]` is less than 0?

**Exercise:** Insert this code, and similar code to handle collisions with the horizontal walls, into your `computeAccelerations` function. To test your code, give atom 0 a nonzero velocity along some diagonal direction. You should then see this atom bounce around inside the box.

**Exercise:** Now add the code to handle collisions between atoms. You'll need a double loop over all pairs of atoms:

```
for i in range(N):
    for j in range(i):
        # compute forces between atoms i and j
        # and add on to the accelerations of both
```

To compute the force components, use the formulas you worked out at the beginning of this project. Be careful with minus signs. Then run your code and enjoy the show!

**Question:** Why does the inner loop (over `j`) run only from 0 up to `i-1`, instead of all the way up to `N-1`?

## Optimizing performance

Ninety percent of the time, you shouldn't worry about writing code that will run as fast as possible. It's much more important to make your code simple and easy for a *human* to read and understand. *Your time is more valuable than the computer's!*

However, your `singleStep` and `computeAccelerations` functions fall in the other ten percent. These functions contain loops that execute the same code thousands upon thousands of times. Any effort that you spend speeding up the code inside these loops will be rewarded in proportion.

**Exercise:** A typical atom in this simulation might have a speed of 1 in natural units. How many units of time will it take this atom to cross from one side of the box to the other (assuming no collisions along the way)? How many calls to `singleStep` are required for such a one-way trip? During this same time, how many times will the code within each of the loops in `singleStep` be executed? How many times will the code within the double loop in `computeAccelerations` be executed? How many times will your code change the position of a `sphere` graphics

object? (Please assume that `N = 100`, `w = 20`, and there are 10 calculation steps per animation frame—even if you've changed these variables in your program. Show your calculations and answers clearly in the space below.)

Here, then, are some tips for optimizing the performance of your MD simulation:

- Don't spend time trying to optimize performance until after you're sure that your program is working correctly.
- Never worry about optimizing code that isn't executed at least thousands of times per second.
- Addition, subtraction, multiplication, and division are fast, but exponentiation (`**`) and function calls (such as `sqrt`) are slow. Therefore, when you compute the Lennard-Jones force, avoid all uses of exponentiation and of function calls. Since only even powers of $r$ appear in the force, you can work with $r^2$ (or better, $1/r^2$) instead of $r$. To square or cube a number, just multiply it by itself. Cube $1/r^2$ to get $1/r^6$, then square that to get $1/r^{12}$. Use intermediate variables where necessary to avoid repeating calculations.
- To speed up the force calculations even more, don't bother to compute the force between two atoms when they're farther than (say) 3 units apart. (Be sure to test whether they're too far apart in a way that avoids calculating a square root.)
- The `singleStep` function makes repeated use of the combinations $dt/2$ and $dt^2/2$, where $dt$ is the time step. So compute these quantities once and for all in your program's initialization section, storing them in global variables.
- A brute-force way to speed up the simulation is to increase $dt$. Naturally, this will also increase the truncation error. Don't try this until later, when you'll have a way of checking whether energy is approximately conserved.

- Graphics can often be a performance bottleneck. In GlowScript, there is overhead associated with changing any of the attributes of a `sphere` (or any other graphics object). That's why I've told you to use the separate variables `x` and `y` for the physics calculations, updating the `ball.pos` values only once per animation frame. On the other hand, you still want the animation to be reasonably smooth if possible, and typically this requires at least 20 or 30 animation frames per second. Try adjusting the number of calculation steps per animation frame, and see what value seems to give the best results.

- If you want to monitor performance quantitatively, VPython provides a `clock` function that returns the current time in seconds. Call it twice and subtract the values to determine how much time passed in between calls. (Measuring the performance in this way is *optional* for this project.)

- Make sure you're running your code from a well-optimized browser. As of this writing, both Chrome and Firefox are quite speedy, while Safari, Internet Explorer, and Edge are significantly slower.

**Exercise:** Spend some time optimizing the performance of your program, and describe the effects in the space below. Which changes seem to make the most difference? After optimizing, how large can you make `N` and still get reasonably smooth animation?

## GUI controls

Your program still lacks two important features: It doesn't let you control the simulation in any way while it is running, and it doesn't give you any quantitative data to describe what's going on. You could add all sorts of features of both types. Feel free to go beyond what the following instructions require!

**Exercise:** Add a button to pause and resume the simulation, as in some of your earlier projects. Test it to make sure it works.

**Exercise:** Add a pair of buttons to add and remove energy to/from the system. A good way to do this is to multiply or divide all the velocities by 1.1. Again, test these buttons to make sure they work.

**Question:** Describe what happens when you continually remove energy from the system. How do the atoms arrange themselves? (Sketch a typical arrangement.)

**Question:** Describe what happens when you continually add energy to the system.

**Exercise:** Play with your program some more, perhaps trying different numbers of atoms, changing the width of the box, and using the buttons to add and remove energy. Describe at least one other interesting phenomenon that the simulated atoms exhibit.

## Data output

What kind of data should you collect from this simulation? Energy is always a good place to start.

**Exercise:** Add `wtext` objects to your program to display the kinetic energy, potential energy, and total energy. Update these displayed values from your main simulation loop, once for each animation frame. To compute the kinetic energy, write a separate function that adds up the kinetic energies of all the atoms and returns the sum. To compute the potential energy, it's easiest to add a few lines of code to your `computeAccelerations` function, using a global variable to store the result so you can access it from your main loop. Be sure to include both the Lennard-Jones intermolecular potential energy and the "spring" potential energy associated with interactions with the walls ($\frac{1}{2}kx^2$, where $k$ is the spring constant and $x$ is the amount of compression). To be absolutely precise, you should add a small constant to the Lennard-Jones energy to compensate for the fact that you're setting the force to zero beyond a certain cutoff distance. After you insert all this

code, check that the energy values are reasonable. About how much does the total energy fluctuate? (Because the energy can be positive or negative, please describe the fluctuation as an absolute amount, not as a percentage.) What about the kinetic and potential energies? What happens if you increase the value of `dt` to 0.025?

Another variable of interest is the temperature of the system. For a collection of $N$ classical particles, the equipartition theorem tells us that each energy term that is quadratic in a coordinate or velocity (that is, each "degree of freedom") has an *average* energy of $\frac{1}{2}k_BT$, where $k_B$ is Boltzmann's constant and $T$ is the temperature. In two dimensions each molecule has only two translational degrees of freedom ($v_x$ and $v_y$), so the average kinetic energy per molecule should be $2 \cdot \frac{1}{2}k_BT$. Thus, in natural units where $k_B = 1$, the temperature is precisely equal to the average kinetic energy per molecule.

In a macroscopic system with something like $10^{23}$ particles, the average kinetic energy per molecule wouldn't fluctuate measurably over time. In your much smaller system, the kinetic energy fluctuates quite a bit. To get a good value of the temperature, therefore, you need to average not only over all the molecules but also over a fairly long time period. This requires just a little more computation.

**Exercise:** Add code to your program to compute and display the average temperature of the system. You'll need a variable to accumulate the sum of the temperatures computed at many different times, and another variable to count how many values have been added into this sum so far. Increment these variables in your main loop (once per animation frame), then divide to get the average, and display this value using another `wtext` object. You'll need to reset both of these variables whenever energy is added to or removed from the system. Also add a button that manually resets the variables. Test your code and check that the results are reasonable.

In a similar way, you can also compute and display the average pressure of the system. In two dimensions, pressure is defined as force per unit *length*.

**Exercise:** Add code to your `computeAccelerations` function to compute the total outward force exerted on the walls of the box, and hence the instantaneous pressure of the system. Store the result in a global variable, and use that variable in your main simulation loop to compute and display the average pressure over time, just as you did for the temperature. To check that your results are reasonable, recall that in the limit of low density (where the distance between molecules is large compared to their sizes), the pressure is given by the ideal gas law: $P = Nk_BT/V$. (In two dimensions, $V$ is actually an area.) Set `N` and `w` so the density of your system is

reasonably low, add energy until the molecules are behaving like a gas, and compare the pressure of your system to the prediction of the ideal gas law. Show your data and calculations below.

**Exercise:** Add one more button to your program, to print the temperature, pressure, and total energy values (to the screen, separated by tabs) when it is pressed. This will allow you to record data for later analysis. Check that it works.

Congratulations! Your molecular dynamics simulation program is finished. Now would be a good time to clean up the code and add more comments if necessary. Then get ready to use your program for a systematic numerical "experiment." You'll be studying a system of a fixed number of atoms, in a fixed volume, over a wide range of temperatures. The number of atoms should be at least 200, but feel free to use more if your computer is fast enough. The volume should be large enough to give the atoms plenty of space: at least 10 units of volume per atom. Before you start to take data, fine-tune your program and check that everything is working correctly.

**Exercise:** Setting `N` to at least 200 and the volume to at least 10 units per atom, use your simulation to determine the energy and pressure as functions of temperature over a wide range of temperatures, from about 0.0001 to 1.0, with enough intermediate points to produce smooth graphs. Before recording each data point, be sure to let the system equilibrate long enough to give reasonably stable values of the temperature and pressure. Also please make some notes to describe the system's appearance at various temperatures. Copy your program's output into a spreadsheet and use the spreadsheet to plot graphs of $E$ vs. $T$ and $P$ vs. $T$. Print the data and graphs and attach them to your lab report. Discuss these graphs in as much detail as you can, correlating them to your written notes. How does the pressure compare to the prediction of the ideal gas law? How does the system's heat capacity behave in the various temperature regions, and how does this behavior relate to what you learned about heat capacities in introductory physics? (Feel free to write your answers on the graphs themselves, highlighting the interesting features.)

Physics 2300 Name _____

Lab partner _____

# Project 6: Random Processes

Many physical systems, such as the molecules modeled in the previous project, are so chaotic that the motions might as well be random. But in that case, why should we even bother to calculate the actual trajectories? For many purposes we can get away with substituting fictitious "random" behavior for the actual behavior of a system. Randomness is also an intrinsic property of quantum mechanical measurements, for instance, of the time when a nucleus decays.

In this project you will write several short computer programs that use computer-generated "random" numbers to model simple physical systems. Programs that use random numbers to choose among various possible outcomes are called *Monte Carlo simulations*, after the famous European gambling resort.

In this project you will also leave GlowScript behind, and learn to work in a more traditional Python environment. There is no need for 3D graphics, or to draw the modeled systems in physical space at all. Nor will you need to animate the modeled processes, in order to watch them play out in time. Instead you will accumulate data in lists and then plot the data, all at once, using static graphs.

## A traditional Python environment

The traditional way to use Python is not through a web browser but through a set of software tools that you install on your local hard drive. These tools include the Python interpreter itself, a variety of add-on packages, and software for editing, launching, and debugging your programs. The possible combinations of these tools are practically unlimited, and developers are constantly adding to these possibilities, creating new ways to adapt Python to new uses. Python's adaptability is both a blessing and a curse. On one hand, it accounts for much of Python's popularity among scientists. On the other hand, it makes it difficult to communicate with someone whose Python environment is different from your own, and it can make the initial installation and setup process rather daunting.

To mitigate these difficulties, I recommend that you take advantage of a free product called the Anaconda Python distribution, which makes many of the most widely used scientific Python tools available through a single download-install process. You may already be using a computer with the Anaconda distribution installed. To install it on your own computer, point your browser to `https://www.anaconda.com/download/` and follow the instructions. Choose the Python version 3 installer (currently version 3.6), rather than version 2.7 (which is provided only for compatibility with older code). Once the installation process is complete, test it out by opening Anaconda Prompt (on Windows) or Terminal (on MacOS) and typing

`python`. You should see a message confirming that you are using Python version 3 as configured by Anaconda. Type `exit()` to get out of the Python interpreter and back to the system-level prompt.

To create your Python programs you will also need a text editing application. A good programmer's text editor for Windows is Notepad++, which you can download from `https://notepad-plus-plus.org/`. A good programmer's text editor for MacOS is BBEdit, which you can download from `https://www.barebones.com/products/bbedit/`. Many other editing applications will work equally well, so if you already have a favorite programmer's text editor, just use that.

Next you'll want to create a folder on your hard drive to hold your Python programs. After doing that, launch your text editor and use it to create a Python program containing the single line of code `print("Hello, world!")`. Save this program in your newly created folder under the name `Hello.py`. Then go back to your command-line window (Anaconda Prompt or Terminal) and use the `cd` command ("change directory") to navigate to your folder (for example, on Windows, type something like `cd \Users\username\Desktop`). Then type `python Hello.py` to launch your program, and check that you see the printed greeting.

(There are many things that can go wrong during the process that I've just described, but it's hard to describe every possible problem—let alone their solutions—here in print. If you encounter difficulties, just ask your instructor or some other expert for help.)

Now that you're no longer working in the cloud, you'll also need a way to back up your programs. I recommend a simple USB stick, but you could also manually copy your programs to a cloud storage site, or just email them to yourself. Be sure to back up your work at the end of every coding session!

## A traditional Python program

Now that you have some traditional Python infrastructure, let's talk about how your programs will differ from the ones you created in GlowScript.

A trivial difference is that you won't include the line `GlowScript 2.7 VPython` at the top of each program. Instead, just start right off with the usual comments that give the program name, your name, the date, and what the program does.

The next lines of your program will *import* whatever Python packages it needs. In a traditional Python environment you can't do much at all without packages. In this project you'll need three packages:

- `math` for common math functions like `sqrt` and `exp`;
- `random` for Python's pseudo-random number generator; and
- `matplotlib.pyplot` to produce professional-quality graphics.

(Some Python packages are more properly called *modules*, but I'll call them all packages for simplicity.)

Adding to the confusion, Python provides multiple *ways* to import a package. Here are four different ways to import some or all of the `math` package:

```
import math            # all math functions via "math." prefix
import math as m       # all math functions via "m." prefix
from math import *     # all math functions with no prefix
from math import sqrt, exp     # only certain math functions
```

In the first case you would say `math.sqrt(x)` to take the square root of `x`, while in the second case you would say `m.sqrt(x)`. In the third and fourth cases you would just say `sqrt(x)`, as in GlowScript. Although that might seem easiest, it can be dangerous to import too many functions in this way because their names can start conflicting with your own functions, and/or with each other.

Here's the package-importing code that I recommend for the programs you'll write in this project:

```
from math import sqrt, exp, factorial
from random import random
import matplotlib.pyplot as plt
```

The first line should take care of the math functions you need. I'll explain the other two when you're ready to use them, below.

Once you've imported the packages you need, much of your code will look the same as in GlowScript. But you will not be using any VPython features: no boxes, spheres, or cylinders; no vectors; and no rate functions to control the speed of animation loops. (If you ever want to use VPython features from a traditional Python environment, there is a VPython package that provides these features. It's not part of the Anaconda distribution, so you would need to install it as a separate step. Unfortunately, I've found it pretty awkward to use. Fortunately, it is not needed for this project.)

You'll also need to adapt to some restrictions that are present in true Python but not in GlowScript—due to the fact that GlowScript is actually a thin Python veneer on top of JavaScript:

- Standard Python won't automatically convert numbers to strings when you try to combine them with the + symbol, as in `print("x = " + x)`.
- Standard Python doesn't allow the `!` symbol in place of `not`, as in `running = !running`.
- Standard Python won't let you add an element to a list by simply giving it an initial value, as in `x[1] = 0` (if `x[1]` doesn't already exist).
- Standard Python won't let you use an expression as a list index if that expression might not evaluate to an integer, as in `x[i/2]`. But you can say `x[int(i/2)]`.

- Standard Python requires that a function definition appear *above* any code that calls the function.

With these constraints in mind, let's now get on with the project.

## Computer-generated "random" numbers

The `random` package provides a function called simply `random()`, which returns a "random" real number between 0 and 1 each time it is called. Before using these numbers in physics simulations, it's a good idea to get some feel for what they're like.

**Exercise:** Write a short Python program called `RandomTest.py` that prints (to the screen) the values of 20 successive numbers returned by `random()`. Run your program from the Anaconda Prompt or Terminal window as before. Do the numbers appear random to you? How can you tell?

**Question:** If you run your `RandomTest` program a second time, do you get the same sequence of numbers, or a different sequence?

Often you'll want random numbers that span a range other than 0 to 1; a common need is to choose a random integer within a certain range. Although the `random` package provides a separate function for this purpose, it's often easier to just multiply the result of `random()` by $n$ and round down using `int()`, to obtain a random integer between 0 and $n - 1$, inclusive. (The `random()` function never returns exactly 1, so when you multiply by $n$ and round down, you'll never get $n$.)

**Exercise:** Add code to your `RandomTest` program (without deleting the code that's already there) to obtain and print the values of 50 random numbers (digits) in the range 0 to 9 inclusive. Count the number of times each digit occurs, and write the results of your counts below.

Another common need is to generate random points in some region of space, in two or more dimensions. To try this out in two dimensions, you can plot the points using the following code:

```
plt.plot(x, y, marker="o", markersize=2, color="red",
            linestyle="None")
plt.show()
```

(I'm assuming that you've already said `import matplotlib.pyplot as plt`.) Here `x` and `y` are *lists* of the points to plot, which you must build in advance; the `matplotlib` package is intended for static graphs, not for graphs that you build up, point by point, as a simulation progresses. By default it will connect the points with straight line segments, but in the code above I've overridden this behavior and instead told it to use red circles with a width of two printers' points.

**Exercise:** Add code to your `RandomTest` program (without deleting the code that's already there) to generate 1000 random real-number $(x, y)$ pairs, with each random coordinate ranging from 0 to 10, and display them using the graphics code given above. (Do *not* round the coordinates to integer values.) Temporarily try omitting the `linestyle="None"` option, and try changing the size and color of the markers. Also try replacing the marker shape code (`o`) with some of the following: `s v ^ D x +`. (To see some of the shapes you'll need to increase the size.)

Besides the plotting options that you just explored, `matplotlib` provides a host of separate functions for adjusting the overall appearance of a graph. Here are several to try:

```
plt.axis("scaled")
plt.xlim(0,10)
plt.ylim(0,10)
plt.xticks(range(11))
plt.yticks(range(11))
plt.grid()
plt.title("1000 random points")
plt.xlabel("x")
plt.ylabel("y")
```

The first of these options forces the scales of the $x$ and $y$ axes to be the same. The next two lines remove the default buffer region around the edges. The rest, I hope, are reasonably self-explanatory. To see a complete list of `matplotlib` functions and options, you can look at the documentation at `https://matplotlib.org/api/pyplot_summary.html`. Just try not to be too horrified by the complexity, or by the way that whoever wrote the documentation assumes that you're already an expert. Poorly written documentation is disturbingly common for Python packages.

**Exercise:** Insert all of the code lines shown above, between the calls to `plt.plot` and `plt.show`. When you are happy with the appearance of the graph, use the

interactive controls to save it, then open the saved image in another application and print it. How many points, on average, do you expect to appear in each of the 100 grid squares of your graph? What are the approximate maximum and minimum numbers of points that actually appear within the squares? (Please circle the corresponding squares on your printout.)

Now that you've gotten a feel for what the `random()` function does, here's the bad news: The numbers that it returns are not actually random. In fact, each of these numbers is computed from the previous one in a completely deterministic way. (The computer's clock is used to get the sequence started, so you get a different sequence each time you run the program. If you ever want to get the same sequence every time, you can do so using the `random.seed` function.) But the function used to generate each number from the last is chaotic, so that even a tiny change in the input produces a substantial change in the output. The function also has the property that over the long term, it generates numbers that are distributed evenly throughout the interval from 0 to 1. But the function is not guaranteed to return numbers that are sufficiently random for all purposes. In particular, despite the appearance of your plot, this function is not guaranteed to generate evenly distributed points in higher-dimensional spaces. In recognition of these imperfections, computer-generated "random" numbers are often called *pseudo-random numbers*.

In this course we'll ignore this potential pitfall and assume that the numbers returned by `Math.random` are sufficiently random for our purposes. For research-quality work, however, you should always make sure your pseudo-random number generator is adequate for the task at hand. For an excellent discussion of this problem and several examples of much better generators (coded in C++), I recommend the book *Numerical Recipes* by Press, *et al.*

## Expansion of a gas

Consider the situation shown in Figure 6.1: A box is divided into two sections of equal size, separated by a partition. On one side of the partition is a gas of $n$ molecules; on the other side is a vacuum. We then puncture the partition and allow the molecules to pass from one side to the other. What happens to the number of molecules on each side as time passes?

To answer this question we *could* write a full-blown molecular dynamics simulation. Or we could just recognize that for our purposes the motions will be essentially random, and say that each of the $n$ molecules has an equal probability of switching sides during any short time interval.

**Exercise:** Write a program called `TwoBoxes.py` to simulate the behavior of this system. Use the variables `n` for the total number of molecules and `nLeft` for the

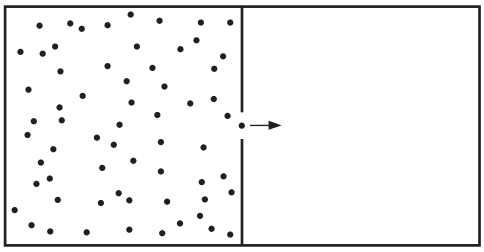


Figure 6.1: All the molecules start on the left side of the partition. What happens when we puncture the partition and let the molecules travel back and forth?

number that are currently on the left side of the box. In each step of "time," choose a molecule at random (by generating a suitable random number) and move it to the other side of the box (by increasing or decreasing `nLeft` by 1). To determine which side your randomly chosen molecule is on, you can simply assume that the first `nLeft` molecules are on the left and the rest are on the right; then there is no need to create any lists to represent the molecules. The output of your program should be a plot of `nLeft` vs. "time." This means that you'll need to build lists of the values to plot; you might call them `tList` and `nLeftList`. Adjust the appearance of the plot appropriately, being sure give it a suitable title and labels on both axes. Turn in a printout of this plot for `n` = 1000.

**Question:** Why did I put the word "time" in quotes? Can you use this simulation to determine the amount of time (in seconds) before about half the molecules will be on each side of the box?

Notice from your simulation that even after roughly half the molecules are on each side of the box, the number on the left side fluctuates significantly. It's interesting to study these fluctuations in a little more detail.

**Exercise:** To study fluctuations around equilibrium, modify your simulation to start with exactly half of the molecules on each side. Run the simulation for `n` equal to 10, 100, and 1000, and in each case, estimate (from the graph) the typical amount by which `nLeft` fluctuates away from `n/2`. Write down these estimated numbers and discuss them briefly. (There is no need to print these graphs.)

**Exercise:** Comment-out the plotting code in your `TwoBoxes` program, and modify it to instead create a histogram plot, with `nLeft` on the horizontal axis and the number of times that that value of `nLeft` occurs on the vertical axis. To store the histogram you'll need a list whose index runs over all possible values of `nLeft`. You can create this list, and initialize all its values to zero, with a statement like `hist = [0] * (n+1)`. After each step of the simulation, add 1 to the appropriate element of this list. It's customary to display the histogram as a bar graph; you can do this by using the `plt.bar` function instead of `plt.plot`. Be sure to give the plot a title and to label both axes. Run your simulation for `n` equal to 10, 100, and 1000, and briefly describe the appearance of the histogram plots. (Don't print any of these plots yet.)

If you've studied a little probability theory, you may have already realized that over the long term, the probabilities of the various possible `nLeft` values should be given by the *binomial distribution*:

$$\text{Probability} = \frac{1}{2^n}\frac{n!}{(n_{\text{left}}!)(n-n_{\text{left}})!} = \frac{1}{2^n}\frac{n!}{(n_{\text{left}}!)(n_{\text{right}}!)}. \tag{6.1}$$

For a sufficiently long run, therefore, the values in your histogram should be approximately equal to this formula times the number of steps in the simulation.

**Exercise:** Add a function definition to your program to compute and return the value of the binomial distribution, as a function of `n` and `nLeft`. Use Python's built-in `factorial` function, which is part of the `math` package. Test your binomial function in a couple of simple cases, and write the results below.

**Exercise:** Now use your binomial distribution function to plot the "theoretical" (long-term average) distribution of `nLeft` values on your histogram plot. Use `plt.plot` for the theoretical distribution, drawing it as a continuous line (no markers), in a color that contrasts with the histogram bars. To add a legend to the graph, insert

```
label="Monte Carlo results"
```

as a parameter in the `plt.bar` function call, and insert a similar parameter into the `plt.plot` function call. Then, before calling `plt.show`, insert the function

call `plt.legend()`. Run the simulation for `n` = 10 and 100, with enough "time" steps to produce reasonably good agreement between the actual and theoretical distributions. Print the plot for `n` = 100.

### Random walks

Next let us consider the erratic motion of a single microscopic particle. The particle could be a gas molecule, a dust grain suspended in a fluid, or a conduction electron in a copper wire. In all these cases the particle collides frequently with neighboring particles and therefore moves back and forth in a way that appears mostly or entirely random. Such motion is referred to as a *random walk*.

The simplest example of a random walk is in one dimension, with steps all the same size, each step equally likely to be one way or the other. This example is mathematically similar to the previous simulation of a gas in two boxes.

**Exercise:** Write a new Python program called `RandomWalk.py` to model a random walk in the $x$ direction. During each time step the particle should move one unit of distance, with a 50-50 chance of moving in the positive or negative direction. Start the particle at $x = 0$. Plot a graph of the particle's position as a function of time for some fixed number of steps, then repeat the simulation several (20 or more) times over, plotting all the results on the same graph. (You can do this by simply making multiple calls to `plt.plot`, all before calling `plt.show`.) Describe the results in the space below, and turn in a printed graph from a typical run of the program. As you increase the number of steps in the walk, what happens to the typical net distance traveled?

**Exercise:** To quantify your answer to the previous question, modify your program to calculate the root-mean-square (rms) net distance traveled by all of your random walkers. This is the square root of the average of the squares of the final positions. After calculating this quantity, simply print it to the screen. Run your program for several different values of the number of steps in the walk, and record the results below. Can you guess an approximate formula for the rms displacement as a function of the number of steps?

**Question:** Why not simply compute the *average* net displacement of the random walkers, rather than the rms displacement?

## Nuclear decay

A classic example of random behavior is nuclear decay. Each radioactive isotope has a certain intrinsic probability of decaying per unit time. As far as we can tell, the time when any particular nucleus decays is truly random.

**Exercise:** Write a new Python program called `Decay.py` to model the decay of a collection of $n$ radioactive nuclei of the same isotope. Let the probability of each nucleus decaying per unit time be 0.001. During each time step, for each remaining nucleus, generate a random number to determine whether that nucleus decays. Once you have calculated how many nuclei decay during a given time step, subtract that number from $n$ and then repeat. Run the simulation long enough for at least 90% of the nuclei to decay. The output of the program should be a graph of $n$ vs. time, labeled as usual. Run your program with $n$ initially equal to 100, and keep a printout of the graph. Repeat for $n = 10,000$.

**Exercise:** Use one of your graphs to determine the approximate half-life of this isotope, that is, the average time for half of the nuclei to decay. Mark this value on the graph and write the result below.

**Question:** When $n = 10,000$, how many nuclei do you expect to decay (on average) during the very first time interval of this simulation? Explain.

**Exercise:** Write a new simulation program (call it `DecayDist.py`) to answer the previous question in more detail. The program should simulate only the very first time interval of the decay of 10,000 nuclei, where each nucleus has a probability to decay of 0.001. However, the program should repeat this one-time-interval "experiment" a hundred or more times, and plot a histogram of the results, so you can see in detail how the actual number of decays tends to fluctuate around the average expected number.

After a sufficiently large number of trials, the histogram plotted by this program should take on a well-defined shape. And as you might guess, there is a fairly simple formula for this shape. If the average expected number of decays is $\lambda$, then the

probability of actually getting $k$ decays is given by the *Poisson distribution*:

$$\text{Probability} = \frac{1}{k!}\,\lambda^k\,e^{-\lambda}. \tag{6.2}$$

**Exercise:** Using pencil and paper, in the space below, sum the Poisson distribution over all possible $k$ values, and interpret the result. To carry out the sum, you'll need to know the so-called Taylor series for the function $e^x$, so look that up if necessary.

**Exercise:** Add a function to your `DecayDist` program to compute the Poisson distribution. (The name `lambda` has a special meaning in Python, so you'll have to use a different variable name.) Using this function, plot the "theoretical" shape of your histogram on the same graph as the actual histogram, using a legend to label the plot as before. Once everything works, print the graph and turn it in with your lab report.

Congratulations! The computer programs for this project are now complete. Be sure to check that all of your code is clearly written and adequately commented. Turn in all five programs (`RandomTest`, `TwoBoxes`, `RandomWalk`, `Decay`, and `DecayDist`) as attachments to an email message to your instructor. Then answer the following question, and turn in this lab report with the printed graphs attached at the end, in order.

**Question:** Briefly summarize how you worked with your lab partner on this project. How did you divide the work, and what fraction of the work did you do yourself? Any other comments?

[This page intentionally left blank.]

# Project 7: Final Project

I hope that by now you are anxious to move beyond "canned" projects and apply what you have learned in this course to something more creative. This final project is your chance to do just that: You will design and carry out a project of your choice in computing applied to science. Your project should satisfy the following criteria:

- It should be comparable in size and difficulty to any one of the other scientific projects you've done in this course (Projects 2 through 6).
- It should entail writing at least 200 lines of original computer code. This could be a single new program, or several shorter programs, or a significant enhancement to one or more of the programs you've already written.
- It should explore a scientific question. This could be another topic in the physics of motion, forces, and energy, or a topic from some other branch of physics, or a topic from some other field in the natural sciences (biology, earth science, etc.).
- The project's main goal(s) should be something that you could not feasibly accomplish without a computer. That is, the project should use computation in a nontrivial way.
- You should put significant effort into analysis and interpretation—not merely into the computer code. (A magnificent piece of code is not by itself an acceptable project.)
- The "deliverables" will be one or more original computer programs, a typewritten paper fully describing the project, and a very brief presentation of the project to the class (described in more detail below).
- You will carry out your final project individually, not with a lab partner.

Although there are unlimited possibilities for projects that meet all these criteria, there are also many possible pitfalls: ideas for projects that meet most of the criteria, but critically fail to meet one or more of them. So please read through the list again, and make sure you understand all the criteria, before you set your heart on a particular project idea. All project topics must be approved by your instructor before you invest any significant time working on them.

You may be able to come up with a great final project topic merely by thinking creatively. Alternatively, feel free to consult textbooks (such as those listed at the end of the Preface to this manual) and other sources for ideas.

Although you should use the Python language for your coding if possible, you may use other computing languages or environments if you have a good reason to, provided that you obtain your instructor's permission in advance.

## Presentation

Please prepare a 5- to 10-minute presentation (depending on the available time) of the highlights of your project, for the benefit of your classmates. Plan to use your classroom's projection system to present the visual components of your project—either running your simulation(s) live, or showing a few PowerPoint slides, or both. (There won't be time to write anything on the board.)

Because of the severe time constraint, your presentation will not be able to cover everything you did. So think about which parts of your work will be most interesting to your audience, and focus on those. Visual presentations are usually more effective than long passages of text or code.

Whether you are using someone else's computer or your own, it is critical that you test the equipment in advance to make sure there will not be technical difficulties with the computer and projection system.

All students are expected to attend all presentations, and to act respectfully toward each speaker. Questions are encouraged to the extent that time permits. If you are not used to public speaking that's just fine; no points will be deducted for being nervous during your presentation!

## Paper

Your paper should be a more complete description of your project. The idea here is to describe your project in enough detail that a classmate, reading your paper, could understand what you've done and reproduce it without too much difficulty.

I suggest dividing the paper into three main parts:

1. An introduction, in which you explain the goals and motivations of your project. What question(s) did you try to answer, or what problem(s) did you try to solve? Why is this project interesting?

2. A description of how you carried out the project. Where did you begin? What were the main obstacles, and how did you overcome them? This will probably be the longest part of your paper, but try to hold your reader's interest by focusing on the most important issues that arose, rather than methodically describing everything, step by step. You may or may not find it appropriate to include some short code fragments, but try to avoid presenting lengthy amounts of code.

3. Your results. Try to present these as visually as you can, with plenty of illustrations. Short tables of numerical results are also sometimes appropriate.

Besides these three main sections, most papers will also include some concluding remarks at the end (but don't merely repeat what you've already said), and a list of the references that you used.

There is no mandatory format for citing references, but try to be consistent and be sure to include the author's name, title, publisher, and date, unless some of this information is not available to you. For online references, also include the exact URL. If you used only a small part of a lengthy reference, please include enough specifics (chapter, section, page, etc.) to direct the reader to the part that you actually used.

But merely citing references at the end of your paper is not enough. Your paper should also be clear at every stage about what sources you used for each piece of important information, how you used those sources, and what creative elements you provided yourself.

In some cases it may be impractical to present all the theory behind your project in a self-contained way, so you'll need to summarize what you can and refer the reader to a reference that provides the details.

Again, you should pretend that you're writing for a classmate. There's no need to review material that everyone in this class has learned earlier, but you should not assume that the reader knows anything about your specific project. Explain things at the same level of detail that you would want to see if you were reading a classmate's paper about a project unfamiliar to you.

Your paper should be written in your best English, with correct spelling, grammar, and punctuation, organized into good sentences and paragraphs. Please write in whatever tone seems natural to you, and avoid excessive formality; it's fine to write in first person. If you need help with your writing, it is your responsibility to obtain that help in a timely manner.

There is no rigid rule for the length of your paper, but if you're finding that it is less than about five double-spaced pages (including illustrations), then you probably haven't explained your project in enough detail. If it is longer than ten pages, then you may be including too much detail (or making illustrations unnecessarily large).