2019

VPython for Introductory Mechanics: Complete Version

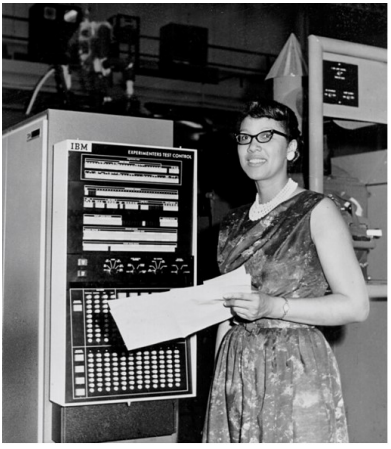
Windsor A. Morgan
Dickinson College

Lars Q. English
Dickinson College

VPython for Introductory Mechanics

Incorporating numerical simulations in the introductory physics curriculum

W.A. Morgan and L.Q. English
Department of Physics and Astronomy
Dickinson College



On the front cover: the large photograph is of Melba Roy Mouton.
She calculated artificial satellite orbits as a Head Computer
Programmer for NASA.

Left Cover Image source: *NASA on the Commons*
Right Cover Image source: *https://pxhere.com/en/photo/1199802*

Contents

Preface

Many physics departments nationally and internationally have moved to incorporate more computation into their curriculum. In many cases this is accomplished by offering an upper-level elective in computational physics. Frequently, computational exercises are also incorporated in upper-level course work. Less often do we see substantial integration of coding in the first semester of physics. One exception is the *Matter and Interactions* curriculum developed at North Carolina State University.[1]

In this book, we describe a series of group exercises that we incorporated into our first course for majors. The calculus-based introductory sequence at Dickinson College is taught in the *Workshop Physics* mode which emphasizes inquiry-based exploration in groups. Students work through exercises in groups of four, and these mostly involve guided observations, measurements and experiments. In the fall semester of 2018, we started supplementing these with numerical projects, and this book grew out of that experience.

In the following chapters, we describe a series of computational exercises that can be easily woven into a project-centered course format. Alternatively, these exercises could also be included into the laboratory part in a more traditional course. The sequence of topics was meant to follow a typical calculus-based intro-mechanics course, as well as an Advanced Placement Physics high school class. (It does not, however, mesh well with alternative paradigms, like the energy-first approach.) For instance, we start with graphing motion using the kinematics equations, then move on to projectile motion, before delving into *dynamics* and the simulation of motion using physical law. The next-to-last chapter is an exploration of rotational motion, torque and angular momentum conservation, topics usually at the end of a typical course. The last chapter considers the "real-world"

[1] R. Chabay, B. Sherwood, *Matter and Interactions* (John Wiley & Sons, 2015)

example of gravitational assist in space travel.

One important precondition for such an endeavor is that the programming language not get in the way of the physics. For us, it was not very important for students to become proficient in coding. In fact, we chose to provide the students with a scaffold by giving them an initial template for almost all exercises, and students were then asked to modify the code to complete the activities. In the same spirit, we tried to keep the code we gave students as short as possible, while encouraging them to embellish as they saw fit.

We chose VPython for a number of reasons. First, it is convenient that it lives online. Nothing has to be installed locally. Secondly, its well-known strengths in 3D visual rendering is obviously a major benefit in a first course in classical mechanics. Finally, its basis in the Python programming language allows students to get some early exposure to a language that is becoming ever more widely used in scientific computation.

L.Q. English and W.A. Morgan
Carlisle, Pennsylvania, August 2019

Introduction

This book is intended to supplement the course materials of a first-semester introductory course in physics by adding a computational dimension into the course content through the use of VPython - a "visual" extension of the *Python* programming language that incorporates 3D animation and vector math. The book is not designed to replace the regular course textbook. Since it is intended to be used in conjunction with other material presented in the course, one of the goals was to make its integration into the course flow as seamless as possible. To this end, we have organized the topics and exercises so that they should more or less follow the traditional sequencing of topics encountered in most introductory mechanics courses.

The exact order of topics was chosen to enable its easy integration into our *Workshop Physics* curriculum.[1] While the pedagogical approach, format and presentation of Workshop Physics is very different from the more standard lecture-based course, its content selection is fairly conventional. Thus, we believe that this presentation should also work well in conjunction with a more traditional introductory mechanics course (either at the senior high-school or first-year college level). Furthermore, swapping neighboring chapters or omitting certain sections should create only minimal disruption.

Having said that, introducing computation in the first semester does allow for some curricular innovation, such as in the form of inclusion of types of problems that are not typically encountered otherwise. In particular, it allows students to examine problems that have no closed-form analytical solutions, to engage in more creative and open-ended problems,[2] and to practice physical modeling. One example we include in this book has to do with the law of universal

[1] *Workshop Physics* was developed at Dickinson College, principally by P. Laws, as an inquiry-based, active-learning curriculum for introductory physics

[2] E.F. Redish and J.M. Wilson, "Student programming in the introductory physics course", *Am. J. Phys.* **61**, 222 (1993).

gravitation. After a sequence of exercises where students discover Keplerian orbits, we ask them to explore what would happen to planetary orbits if gravity were instead a "1/r" kind of force. Such a question would be quite difficult to answer analytically, and would require an upper-level arsenal of mathematical tools, but intro students can tweak the code very easily to explore this scenario.

Thus, the use of VPython has the pedagogical virtue of empowering students to engage in more physical modeling, and it enables them to better visualize processes, both of which in turn strengthen students' conceptual understanding.[3] Finally, a very straightforward benefit for introductory students is that they are exposed to a language, Python, that is fast becoming the premier tool for scientific computation.

The structure of the book is as follows. We start in Chapter 2 with some preliminary exercises to get students familiar with the interface and basic coding in VPython. In Chapter 3, we then turn our attention to one-dimensional motion and motion diagrams. Quite a lot of emphasis is placed in Workshop Physics, as well as *RealTime Physics*,[4] on describing basic motion in multiple representations: verbally, in graphical form, and with mathematical equations. Physics education research has shown that real conceptual and problem-solving progress occurs when a student can move fluidly between these various representations of motion.[5] The idea in this chapter is to bolster students' comprehension of position- and velocity-graphs, which are one layer removed in abstraction from the literal description of the motion, by having them program an object performing a simple kind of motion on the screen and then

[3] R. Chabay, B. Sherwood, "Computational Physics in the introductory calculus-based course", *Am. J. Phys.* **76**, 307 (2008).

[4] D. Sokoloff, R. Thornton, P. Laws, *RealTime Physics - Active Learning Laboratories* (John Wiley & Sons, 2012).

[5] A. Van Heuvelen, "Learning to think like a Physicist: A review of research-based instructional strategies," *Am. J. Phys.* **59**, 891 (1991).
L.C. McDermott, "A View from Physics" in *Toward a Scientific Practice of Science Education* (Lawrence Erlbaum Assoc., 1990).

having VPython generate the corresponding motion graphs. In a sense, it is the computational analogue to the common laboratory exercise of using a motion sensor and walking in front of it.

In Chapter 4 we assume the course has by now advanced to a discussion of average and instantaneous velocity and acceleration. (In a traditional intro course, this may come very quickly.) We use some of these ideas to introduce the basic Euler method at the heart of all numerical simulation encountered here.

In Chapter 5, we return briefly to the realm of plotting formulas, as the course is now at the point of exploring projectile motion, i.e., two-dimensional motion under constant acceleration. This topic is naturally suitable for adding a computational facet. Exact analytical results (such as the range formula) can be easily tested numerically, for instance, as outlined in the exercise section of this chapter. Then, additional complications can be added (such as a sloped ground). We end by having students code a "hit-the-target" game.

Chapter 6 in many ways strikes at the heart of mechanics. We are finally ready to simulate Newton's laws, and the full power of computational physics is revealed for the first time. We can see the kinds of motion that Newton's laws actually give rise to, given the forces acting on a particle and the particle's initial state. We specifically highlight three important problems in order of increasing difficulty: the 1D mass-spring problem, projectiles traveling through air, and the orbits of planets and comets. What makes this approach so potent in the context of an introductory course is that the analytical solutions are too mathematically challenging to derive and therefore remain beyond reach for this audience. Yet, the numerical pathway is accessible.

For Chapter 7, it is assumed that the course has now reached a discussion of momentum and energy. We chose the binary star system

as a vehicle for introducing the concept of center-of-mass and to tie it to momentum conservation. We then return to the mass-spring problem to illustrate the conservation of mechanical energy in this system. The Rutherford experiment can be skipped, but it provides a nice preview to electrostatics.

Maybe unsurprisingly, we now follow up with rotational motion and angular momentum, which often represents the last material covered in the semester. We end with what we believe to be two engaging final examples - the gravitational "slingshot" and chaotic orbits - as a kind of capstone project for students to explore.

Lastly, before we get started, all the VPython codes featured in this book can be accessed online at
https://scholar.dickinson.edu/vpythonphysics/ .

Getting Started with VPython

2.1 "Hello."

VPython is a language that is pretty easy to use, once you get used to it. Python is becoming widely used by scientists around the world, and VPython is a visual implementation of it. It will help you visualize the physics you will be learning.[1]

To learn the basics of programming in VPython, the first thing we will do is one of the simplest – having the program print the phrase "Hello, world!" – a traditional first program.

To do so, we will be in the Glowscript-VPython programming environment – an environment that is easy to use and understand. Here is what you do:

1. Launch a web browser (try Google Chrome), and go to the site www.glowscript.org . The webpage will look like this:

2. Sign in with a 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.

[1] See also D. Schroeder, "Physics Simulations in Python" (2018).

Figure 2.1: The Glowscript webpage.

6. You will now see an editing space that is blank except for the line "GlowScript 2.8 VPython" (possibly with a different version number). Click in the white space below that line and type the following, verbatim:

```
print("Hello, world!")
```

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, world!"

8. Click on "Edit this program" and replace "Hello, world!" with something else. Run the program. You should get similar results.

If the program didn't work, check the spelling and capitalization and punctuation. Make sure that you typed the line precisely as shown above.

GlowScript-VPython is a programming environment that stores your programs in the cloud. The interface is very spare – however, there should be no difficulty just starting it up, making new programs, saving them, and managing them.

If you click "Edit this program", you can see the program you just typed. The first line, which is the same for all GlowScript programs, tells the system the language (VPython) and version (2.8) that is to be used. You must have this line, and you should always just leave it alone.

The next line tells the computer to call the print function and passes to it a string of characters. Don't worry that you don't know what the italicized words mean – we'll explain them here:

- The *function* is a pre-defined task that does something – in this case, print to the screen. Just as you don't need to know how a square-root key on a calculator works, a function does its job on the information *passed* to it. This information is called the *argument* or the *parameter* (they are slightly different, but there is not much of a difference). Here the argument is the *string*.
- The *string* is text that is either single- or double-quoted.
- To tell the program what to do it and when, the function is *called*, which is done in the act of stating the function (here, "print").

2.2 Box – Your First Shape

Now, let's do something a bit more exciting.

On a new line below your print command, type the following command:

```
box()
```

Here you're calling a function called `box`, and passing it no arguments at all (but notice that you still need the parentheses). Run the program again, and you should see a black rectangular area (called a canvas) containing a light gray square (the box), such as one shown in Figure 2.2. Again, this function is doing a lot of work "under the hood", but you do not need to concern yourself about how.

The box that you see on the screen is actually three-dimensional on the screen. You are viewing it closely from one side, so it appears as

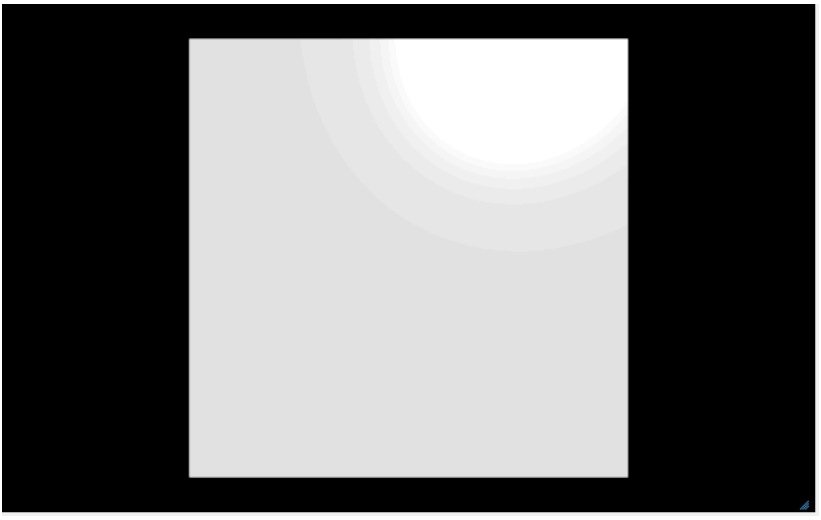


Figure 2.2: A Simple Box

a square. To change the perspective, you can do three things:

- **Rotate**: Right-click button on your mouse and drag the box one way or another, or, if you don't have a right mouse button (such as on a Macintosh), 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.
- **Pan**: Hold down the shift key while you drag using the mouse or trackpad.

You can change the attributes of your box by passing some different parameters to the box function. Try typing this to replace the original box line:

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

Here you're providing *three* parameters, separated by commas, and you're identifying them by their names, which means you can provide them in *any* order. The *pos* parameter specifies the position of the *center* of the box in Cartesian coordinates; the size parameter specifies its dimensions (so here, it is not a cube; 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. We will be learning more about this powerful function soon. This function in turn takes three parameters, x, y, and z, which are usually given in that order. When first presented and before any rotation, the x direction points to the right; the y direction points up; and the z direction points directly outward, toward you (or "out of the screen").

You can learn more about other colors by clicking the **Help** link at the upper-right corner of the window. It might be beneficial to open the help page in a new browser tab. Then, from the second drop-down menu in the left sidebar, choose "Color/Opacity".

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

Exercise: Create at least two more boxes, so you'll have a total of at least three, each with different positions, shapes, 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 with that range. Make sure to "Grab" (a MacOS program) a screenshot of what you did.

2.3 Another shape

Fortunately, VPython provides functions for creating many different 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!

2.4 Comments

Sometimes you want to have some comments to remind you what a certain line or area of a program is doing. All you need to do is put a # sign in front of the comment. Then VPython simply ignores everything following the # sign on that line. (It is very easy to "comment out" a line of code you want to **not** run, sometimes as a diagnostic. Then you can remove the # sign if you want the program to execute that command).

Exercise: Put a multi-line comment at the top o your program right after the line "GlowScript 2.8 VPython", to indicate the name of your program, your own name, and the day 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.)

Moving Objects Using Formulas

In this activity, you will be doing four simple exercises. In three of them, a ball will move with either constant velocity or in in the presence of a constant force. In the last, you will move a planet (just a large ball!) around the Sun.

3.1 Motion of a Ball with Constant Velocity

Examine the program provided for you in Figure 3.1. Type it in. Please note that the indentations are important (as is capitalization, usually).

```
1  GlowScript 2.8 VPython
2  # Defining the Object and Constant Velocity
3  obj=sphere(pos=vector(-1,0,0),radius=0.1,color=color.red, make_trail=True)
4
5  # Setting initial conditions and step size, dt
6  t=0
7  dt=0.05
8  x0=-1
9  v0=1.0
10
11 # This is main part - the loop
12 while t<2:
13     rate(10)
14     x=x0+v0*t
15     obj.pos=vector(x,0,0)
16     t=t+dt
```

Figure 3.1: Motion of a ball with constant velocity

What is each line doing? You can probably figure it out using the commented-out lines. Now run the program. What is happening?

You probably found that this program is having a sphere move horizontally at a constant velocity. The command in line 14:

```
x = x0 +v*t
```

is telling the program to take the original value of x, x0, and to add a constant velocity times the time elapsed t. This works, but we will see in the next chapter that it is not the most elegant way of updating the value of x.

3.2 Motion of a ball with constant velocity with a plot of position vs. time

Looking at the motion of the ball is fine, but plotting the values of position, velocity, or acceleration versus time is a way to visualize what is happening.

Take the first program, copy it with a new name, and add commands to plot the ball's position with respect to time. To do this, you must set up a graph and declare its width and height:

Your program should look something like the one in Figure 3.2:

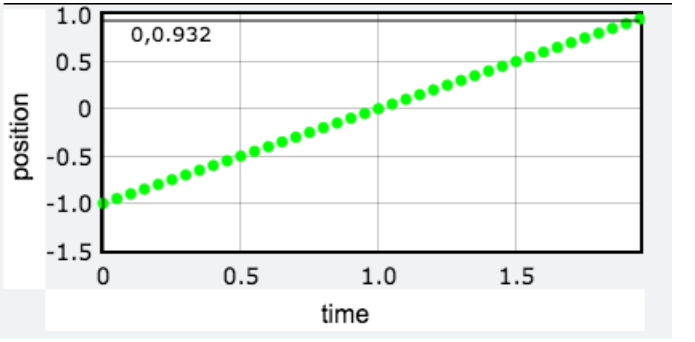
```
1  GlowScript 2.7 VPython
2  # Setting up the Position Graph
3  g1=graph(width=400, height=200,xtitle='time', ytitle='position')
4  xDots=gdots(color=color.green, graph=g1)
5
6  # Defining the Object
7  obj=sphere(pos=vector(-1,0,0),radius=0.1,color=color.red, make_trail=True)
8
9  # Setting initial conditions and step size, dt
10 t=0
11 dt=0.05
12 x0=-1
13 v0=1.0
14
15
16 # This is main part - the loop
17 while t<2:
18     rate(10)
19     x=x0+v0*t
20     obj.pos=vector(x,0,0)
21     xDots.plot(t,x)
22     t=t+dt
```

Figure 3.2: Program Illustrating Graphing Commands

The command **graph** in line 3 sets up a graph with width 400 pixels, height 200 pixels, with an x-axis labeled "time" and a y-axis labeled "position". The set-up graph is called "g1" for ease of use. Line 4 uses the command **gdots** to say how the data is to be plotted, and call it "xDots". Here, it says that green dots should be used in the graph called "g1" that was set up in the line before. Finally, line 21 says that "xDots" should be plotted with the values of "t" (for time) for the x-coordinates, and the values of "x" (for position) for the y-coordinates.

Your plot should like what is shown in Figure 3.3. Is this what you would expect? Why?

Figure 3.3: Plot of position versus time for ball with constant velocity.



3.3 Motion of a Ball Being Dropped from Rest Near the Surface of the Earth

Now, let's simulate the motion of a ball being dropped from a height of two meters.

Questions: What has to be done to modify the second program to mimic the ball being dropped? What is different about this ball's velocity compared to the ball's velocity in the second program?

What does the position-time plot look like now? How is it different from the previous plot? Why?

Plot the velocity versus time. How does this compare to the current position versus time plot and the previous position versus time plot?

3.4 Stop and Go

In the previous section you probably discovered that instead of using a constant speed `v0`, you needed to increment the speed with every pass within the `WHILE` loop.

Now let's use a similar idea to make the ball move to the right at a constant speed for 2 seconds, then to have it stop and sit there for 2 second, and then to have it move to the left at a constant speed for 2 seconds.

Discuss a possible strategy with your partners. You may find that there are multiple ways in which you could accomplish this objective. One way would be to have three different WHILE loops - one after the other. However, perhaps a more elegant approach uses the **if-then-else** syntax - a core syntax structure of any programming language. In VPython, it gets implemented in a very intuitive way:

If ("Condition to be tested"):
  "Execute these commands"
Else:
  "Execute those commands"

Here "Condition to be tested" should be something like `t<2`, `t>2`, or `t==2`, for example. Notice the double equal signs that are needed because the line should be interpreted as a test of a conditional statement (and not an assignment). Note that the indentations are essential - horizontal alignments signify and enforce structure in any Python program.

Can you incorporate if-statements to make the ball move to the right, stop, and then move to the left without any discontinuities? It may

take you a while to get the third part right.

3.5 Motion of a Planet

In Figure 3.5 examine the program simulating the motion of a planet (the Earth) around the Sun. This is a very simplified depiction; it does *not* take into account gravity or any of the laws of planetary motion. It merely assumes the Earth's orbit is a perfect circle (which in reality is not too far from the truth). In the program, note that there are cosine and sine terms, to define the x and y positions.

```
GlowScript 2.8 VPython
# Circular Planetary Motion
obj=sphere(pos=vector(10,0,0),radius=0.5,color=color.red)
sun=sphere(pos=vector(0,0,0),radius=1.0,color=color.yellow)


# Setting initial conditions and step size, dt
t=0
dt=1
y0=2
v0=0.0
theta=0
thetavenus=0
omega=2*pi/365


# This is main part - the loop
while True:
    rate(100)
    x=10*cos(theta)
    y=10*sin(theta)
    obj.pos=vector(x,y,0)
    theta=omega*t
    t=t+dt
```

Figure 3.4: Simplified Motion of a Planet Around the Sun

The term omega, written as ω, is called the angular velocity (or the frequency). You will see this term later in the course when we discuss rotational motion. Its value in the program will help you to see

how it is defined.

Exercise: Change the parameters of the provided program. How does the executed program change?

Exercise (optional): Define a third sphere and call it "Moon". Use the same approach to make the Moon revolve around the Earth, while the Earth still simultaneously revolves around the Sun. Hint: Vector addition will be very useful here. You have to add something to the position vector of the Earth.

A First Look at Simulating Motion

In the last activity, we managed to move objects on the screen by repeatedly evaluating the position function, x(t), for successively larger time values. This iterative evaluation of the function was done in the WHILE loop.

We were also able to graph the values in a position-time graph.

This approach was quite powerful already, but it also limits what we can do. For example, it would already be quite difficult to make an object move forward, stop for a while and then move backward using this approach, as you probably discovered in the last chapter. Furthermore, this approach also imposes solutions rather than numerically finding them, thus sort of bypassing the underlying physical laws, as we will see later.

For these reasons, we would like to move from a mere evaluation of a function to an actual simulation of a physical process in real time. Today, we will take our first step in the direction of a true numerical simulation of physics.

4.1 The Physics

Let's start with a piece of physics that you already know – the formula for average velocity in one dimension – and then translate it to a statement a computer can understand. So, average velocity is defined as,

$$v_{avg} = \frac{\Delta x}{\Delta t} = \frac{x_2 - x_1}{\Delta t} \tag{4.1}$$

In words, the average velocity over a time interval of motion is defined as the displacement divided by the elapsed time.

As a general rule, computers cannot handle continuous motion. Therefore, in order to make any simulation of real physical motion look *realistic*, we will have to "chop up" the true continuous motion into many, many frames separated by very small time intervals. The situation is not unlike watching a movie where we perceive continuous motion when in reality we are being exposed to a rapid sequence of still images.

The upshot is that we will want to make our time interval Δt really small in Equation (4.1). Over the course of this small time interval we would not expect the velocity to change very much, so a pretty good assumption is that it is simply constant over this interval. This also means that the average velocity is just the same as this constant velocity and we can drop the subscript "avg". If we then solve Equation (4.1) for x_2, we get:

$$x_2 = x_1 + v\Delta t \tag{4.2}$$

Here we assumed that Δt was sufficiently small, which allowed us to omit the subscript *avg*.

Let's describe Equation (4.2) in words: Between two frames separated in time by Δt, we can compute the position of the object in the new frame, x_2, based on the position in the current frame, x_1, and the velocity of the object, v. We simply have to add $v\Delta t$ to x_1.

Now all we have to do is repeat the step in Equation (4.2) over and over again. Every time we advance forward in time by the small

amount Δt. And every time, we replace the starting position x_1 by the ending position x_2 of the previous step. In other words, we want to use Equation (4.1) recursively in order to evolve the position of our object forward in time, step by step.

So far, we have assumed that v in Equation (4.2) would remain constant in each successive step, but this does not have to be the case. In general, we can allow the velocity v to get updated as well with each new iteration. We will see some straightforward examples in the section 4.3. The procedure we have described here is also known in the numerical analysis community as the *Euler Method*.[1]

4.2 The Basic Code

Now that we a taste of the physics involved, how do we translate that idea into code a computer can interpret? Let's take a look at the following few lines of VPython code - see Figure 4.2.

We see that the core of the program is contained in the WHILE loop, as before. Let's start our discussion with the central line, namely LINE 22:

$$x = x + v * dt \tag{4.3}$$

As a mathematical statement this is, of course, nonsense. But we should not interpret the equal symbol in the mathematical sense. It is not an equality. It is an ASSIGNMENT. "x" is a variable that holds a certain number, and whenever you see "x = something" this signals that the value stored in the variable "x" is about to be updated.

So Equation (4.3) should be read in two parts. The expression to

[1] Named after the mathematician and physicist Leonhard Euler.

```
1  GlowScript 2.7 VPython
2
3  # Defining the "graph"
4  g1=graph(width=400, height=250)
5  xDots=gdots(color=color.green, graph=g1)
6
7  # Defining the Object
8  obj=sphere(pos=vector(-1,0,0),radius=0.1,color=color.red)
9
10 # Setting initial conditions and step size, dt
11 t=0
12 dt=0.05
13 x=-3.0
14 v=2.0
15
16 # This is main part - the loop
17 while t<3:
18     rate(10)
19     obj.pos=vector(x,0,0)
20     xDots.plot(t,x)
21     # Updating the position
22     x=x+v*dt
23     t=t+dt
```

Figure 4.1: The basic code

the right of the equal sign is computed first based on the current value of x. Secondly, the "x = " part then assigns the result of that computation to the variable x. The upshot is that x is updated and now holds the new value.

Thus, if we had to translate the statement in Equation (4.3) into a mathematical equation, we would say:

$$x_2 = x_1 + v\Delta t,$$

where x_2 is the new value and x_1 is the old value of position. Note also that "dt" in Eq. 4.3 should not be interpreted as a differential in the Calculus sense but represents a small time interval, defined in LINE 12 as 0.05.

We are now in a position to discuss the entire code. Line 4 and 5 set up a position graph (time on the horizontal axis, position on the

vertical axis). LINE 8 defines the object we want to move around in space – a red sphere of radius 0.1. Lines 11-13 set the initial conditions as well as the size of the small time interval, dt. Within the WHILE look, we keep updating the variable x and then, using this, to update the position of the object using the command in LINE 19: **obj.pos=vector(x,0,0)**. Here **obj** was defined as the sphere, and **pos** is an attribute of the sphere, namely the position (i.e. location) of the sphere's center. This position is a three-dimensional vector which we create using the **vector** command. Finally, the **xDots.plot(t,x)** command in LINE 20 appends the newest data point to our position graph.

The last line, Line 23, updates the time variable. This is necessary in order to test the conditional of the WHILE loop **(t<3)**, as well as for the purpose of plotting the position (and velocity) graph. It does not, however, come into play in calculating the position of the object – an important difference from the previous approach of simply plotting the function **x(t)**.

4.3 Exercises

1.
   - Run this program to make sure it works. Describe what you see.
   - Modify the code so that during the first second, the ball is moving to the right as before, but for the second second it stays still, and during the third second, it returns to the original starting point. Also have VPython generate the corresponding position-time graph.

     Hint: Think about how you could use **if-then** statements to accomplish this task.

2.
- Discuss with your group how you would have to modify the code printed above to make the object accelerate to the right at constant acceleration. *Hint*: Think about the role of "v"!
- Once you verify that the code really does produce an accelerating object on the screen, let's have VPython generate the position-time graph, as well as the velocity-time graph. Ideally, the data would be displayed in two separate graphs. You may have to duplicate and slightly modify some of the lines already appearing in the sample code.

  Take a screenshot of the code, as well as the two graphs to include in your lab notebook.
- Now that you have the position-time graph, verify that it agrees with the first kinematics equation: $x(t) = x_0 + v_0 t + \frac{1}{2}at^2$. You can do this, for instance, by evaluating the formula at a particular time (for the initial velocity and the acceleration that appears in your code). Does this point lie on the the graph?

Visualizing Projectile Motion

In this activity, you will extend what you learned in Chapter 3 about the kinematics equations into two dimensions. These equations describe the motion of a projectile undergoing a constant acceleration.

Usually a projectile, such as a thrown ball or a launched rocket, is propelled at an angle to the ground θ, usually measured from the horizontal. One must take into account the *components* in the horizontal (usually "x") and vertical (usually "y") directions.

5.1 The Physics

The kinematic equations are as follows:

$$x = x_0 + v_{x0}t + \frac{1}{2}a_x t^2; \tag{5.1}$$

$$v_x = v_{x0} + a_x t; \tag{5.2}$$

and

$$v_x^2 = v_{x0}^2 + 2a_x(x - x_0). \tag{5.3}$$

While "x" is used above in Equations (5.1), (5.2), (5.3), they are simply placeholders; one can easily replace them with "y":

$$y = y_0 + v_{y0}t + \frac{1}{2}a_y t^2; \tag{5.4}$$

$$v_y = v_{y0} + a_y t; \tag{5.5}$$

and

$$v_t^2 = v_{y0}^2 + 2a_y(y - y_0). \tag{5.6}$$

Therefore if we have a projectile launched at an angle θ to the ground, then the horizontal component of the velocity is

$$v_x = v\cos\theta \tag{5.7}$$

and the vertical component of the velocity is

$$v_y = v\sin\theta. \tag{5.8}$$

So, if we throw a ball at angle $\theta = 43°$, with a velocity of 10 $\frac{m}{s}$, then the initial x- and y-components of the velocity are (cf. Equations (5.7) and (5.8))

$$v_{x0} = 10\frac{m}{s}\cos 43°$$

and

$$v_{y0} = 10\frac{m}{s}\sin 43°.$$

As for acceleration, if we are talking about locations near the surface of the Earth and ideal conditions (no atmosphere, so therefore no wind), there is no acceleration in the horizontal (x) direction. The only acceleration is the familiar $a_y = -g = -9.8\frac{m}{s}$.

5.2 Exercises

5.2.1 Projectile over Flat Ground

Use Equations (5.1), (5.2), (5.4), (5.5), (5.7), and (5.8) to write VPython code that simulates the flight of a projectile. Make the projectile a sphere of radius 0.1. Assume that there is no air resistance. Assume that we launch the projectile at the origin (0,0), and stop the projectile motion once it hits the ground at y = 0. This means that the terrain is flat. Draw a line (or horizontal plane) that indicates this.

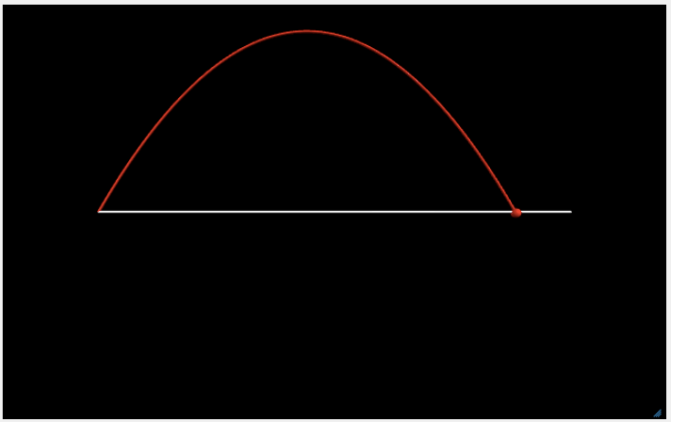


Figure 5.1: A Projectile Trajectory Over a Flat Slope

To accomplish this last condition, use the following `while` loop:

```
while(y>-0.001)
```

Your program should have an output such as depicted in Figure 5.1.

One thing to keep in mind is the built-in unit for angles in VPython is always **radians**. This means that you will need to convert any angle in the argument of a trig-function from degrees to radians.

Now play around with launch angle at constant launch speed. Make a table of the range for each launch angle. For what angle θ do we seem to get the largest range? Is this range consistent with the theoretical formula for the range of a projectile, given by

$$R = \frac{v_0^2 \sin 2\theta}{g}. \tag{5.9}$$

To answer this question, open up Excel, have it compute the theoretical value from Equation (5.9) at regular angle intervals, and then compare the values with your numerical results from VPython simulations.

5.2.2 Projectile over Sloping Ground

Now let's make the ground not level (or flat). Let's have a steady downward slope. Redraw the line (or plane) so that it tilts down with a slope of rise/run = -0.1.

The way we can simulate that in the code is by modifying the WHILE statement. How do you have to do that? Once your group finds a solution, implement it in the code.

What angle now produces the largest range?

Repeat the simulation, but with a terrain that gently slopes up, with a slope of +0.1. What is the best angle now?

5.2.3 Projectile Striking a Target

In addition to the projectile and the (flat) ground, draw in a stationary sphere of radius 0.5. You decide the coordinates of this stationary sphere.

Make the code stop if or when the projectile hits the object, which we will define as anywhere within the volume of the object. If we do not hit the sphere, then make the code stop as before when the ground it hit. Because we are only thinking in two dimensions, in order to hit the sphere we need to just have the projectile get within 0.5 units of the center. How can we do that?

Hint: there is a kind of brute-force approach using the Pythagorean theorem, but there is also a much more elegant method using the two position vectors associated with the projectile and the target sphere. Can you see how you could use the difference of these two vectors?

One thing that will probably need here is the following command:

BREAK

If this command is encountered within a `WHILE`-loop, we immediately leave the loop and continue with statements outside of it. Oftentimes, you will find the `break` command wrapped inside an `IF`-statement all inside a `while`-loop.

Projectile motion is a fairly easy thing to simulate in VPython. In the next chapter, you will see how to make the simulation more realistic by the incorporation of air resistance in the code, with the use of vectors.

5.2.4 Mystery Code

Scan the following VPython code and annotate each line. Discuss what you think this code will do. Make a prediction as to the kind of motion that would be highlighted.

Note that in the VPython environment, to raise a quantity to some power, you do NOT use the carrot-symbol, but instead you use two star-symbols "$**$". Thus, x^2 would be represented in the code as $x**2$.

```
GlowScript 2.7 VPython

graph(width=400, height=250)
xDots=gdots(color=color.green)
redbox=box(pos=vector(4,-1,3),
           size=vector(8,1,6),color=color.red)
ball=sphere(pos=vector(4,50,3),radius=2,color=color.green)

t=0
y0=50
v0=0
t0=0
i=0
while True:
    rate(200)
    t=t+0.01
    y=y0-0.5*9.8*t**2+v0*t
    if(y<0):
        i+=1
        t=0
        y0=0
        v0=20*0.9**i
    ball.pos=vec(4,y,3)
```

Simulating Central-Force Problems

6.1 Stating the general problem and a possible line of attack

In Chapter 4, we learned how we could use an iterative procedure (called the Euler method) to advance the position of an object in small increments based on its current velocity. This allowed us to numerically obtain $x(t)$ given $v(t)$. In other words, we performed a numerical integration: given the known velocity-time graph, we generated the position-time graph.

The task in many physics problems is slightly different, however. Very often, we know the force acting on an object at all points in space, and we would like to somehow calculate the object's trajectory in response to the forces it encounters. How do we do that?

So, let's assume that we know the force that is acting on an object as a function of the object's location. What this means is that no matter where the object happens to be, we can calculate the force that it experiences there. Mathematically speaking, what we are given is the force-function, $\vec{F}(\vec{r})$. This notation communicates that we can evaluate a force vector, $\vec{F}$, by inserting a position vector, $\vec{r}$, into a function. Mathematicians would call such a function a map from R^3 to R^3. A good example are the so-called *central-force problems*, such as the gravitational force a comet feels in the vicinity of the sun.

We also know that the force on the object is related to the acceleration of the object via Newton's second law:

$$\vec{F} = m\vec{a} \tag{6.1}$$

This means that simply dividing the force-function by the object's mass gives us the acceleration function $\vec{a}(\vec{r})$.

Now that we have the acceleration, how do we get the trajectory? Recall that we faced a somewhat similar task in Chapter 4. There we knew the velocity and were able to compute the position. But now we are one step further removed. We know the acceleration, not velocity. What's more, we know the acceleration not as a function of time, but as a function of position.

You may know that in these types of problems, in order to compute the object's trajectory, we must know how the object was initialized. At what position was it released and what velocity did it have at that moment? These two things must be known to us if we are to find the unique trajectory - they are called the *initial conditions*.

So here is the basic idea. Let start from this position and velocity in our code, call them: $\vec{x}_1$ and $\vec{v}_1$. From $\vec{x}_1$ we can compute the force at that location and thus the acceleration $\vec{a}_1$. Using this $\vec{a}_1$, we now update the velocity. Here we make use of the formula for average acceleration,

$$\vec{a}_{avg} = \frac{\Delta\vec{v}}{\Delta t}. \tag{6.2}$$

As the time interval gets very small (and, in the limit, infinitesimal), the average acceleration becomes the instantaneous acceleration, and so,

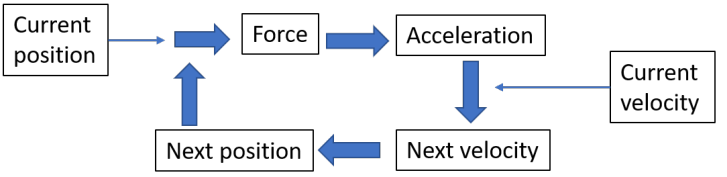
$$\vec{v}_2 = \vec{v}_1 + \vec{a} * \Delta t. \tag{6.3}$$

This completes the second step.

In the third and final step, we now use this new velocity to update the position, using the vector-verson of Equation (4.2), namely:

$$\vec{x}_2 = \vec{x}_1 + \vec{v}_2 * \Delta t. \tag{6.4}$$

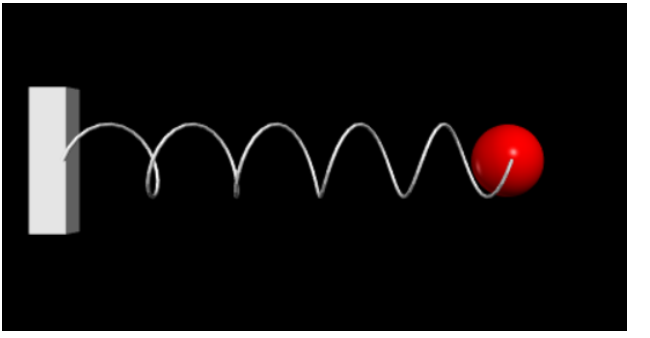
Now that we have computed the new position vector $\vec{x}_2$, we can start the whole procedure over again by first computing the new force and acceleration ($\vec{F}_2, \vec{a}_2$), then the new velocity ($\vec{v}_3$), and then finally the new position ($\vec{x}_3$). Schematically, we are proceeding in steps given by the following flow chart:



In summary, we basically apply the Euler method twice, once for the velocity and then for the position. Notice also that we update the velocity first and then use the new velocity to update position second. It turns out that this order of operation is often superior to the reverse order in terms of computational accuracy. It is called the *Euler-Cromer* method [1].

[1] A. Cromer, Stable solutions using the Euler Approximation, *American Journal of Physics*, **49**, 455 (1981).

6.2 A first example: The mass-spring problem



To illustrate the method, let's get our feet wet with a relatively simple problem - a mass on a horizontal spring. All we have to remember is Hooke's law for ideal springs,

$$F(x) = -kx, \tag{6.5}$$

where k is called the spring constant which corresponds to the stiffness of the spring, and x represents the amount of stretch (if positive) or compression (if negative) of the spring. We should appreciate this formula as the one-dimensional version of $\vec{F}(\vec{r})$ from before. Dividing Equation (6.5) by the mass of the object that we attach to the end of the spring yields the object's acceleration.

Let's look at the basic VPython code, shown in Figure 6.1, to see how the Euler-Cromer method gets implemented in practice here. The basic steps (outlined in the flowchart in the previous section) are contained in lines 24 through 27. Line 28 is not absolutely necessary, but it is included for the purpose of making position and velocity-time graphs.

Notice also how easy it is in VPython to draw a spring-mass setup - we basically select a sphere for the end-mass and a helix for the

```
GlowScript 2.7 VPython

g1=graph(width=400, height=250)
xDots=gdots(color=color.green, graph=g1)
vDots=gdots(color=color.red,graph=g1)
eDots=gdots(color=color.blue, graph=g1)

obj=sphere(pos=vector(-1,0,0),radius=0.5,color=color.red)
spring = helix(pos=vector(-5,0,0), axis=vector(4,0,0), radius=0.5)
wall=box(pos=vector(-5.5,0,0),length=0.5, height=2, width=0.25)

t=0; dt=0.01
x=-1
v=0
k=1; m=1

while t<25:
    rate(200)
    obj.pos=vector(x,0,0)
    spring.axis=vector(x+5,0,0)
    xDots.plot(t,x)
    vDots.plot(t,v)

    F=-k*x
    a=F/m
    v=v+a*dt
    x=x+v*dt
    t=t+dt
```

Figure 6.1: The basic code simulating a mass at the end of a spring

spring. The only thing we have to take care of is to deform the helix in dependence upon the position of the end-mass.

6.3 Exercises

- Run the program in Figure 6.1 and examine the position-time graph. What does the graph look like? What math function does it seem to follow? What about the velocity-time graph?

- What can you say about the relationship between the position-time graph and the velocity-time graph. Is there

a phase difference between them, and if so, approximately what is it?

- Find the place in the code where the initial conditions (x_0 and v_0) are specified. Try running the program with a few different sets of initial conditions. Does the period of oscillation seem to depend on this choice?

- Find the place in the code where system's physical parameters are specified, namely the spring constant and mass. Change one of these parameters at a time, and observe qualitatively how it changes the period of oscillation.

- What if you double (or triple) both the mass and the spring constant? Do the graphs change?

- Are your observations above consistent with the well-known formula for the period of oscillation, T, given below?

$$T = 2\pi\sqrt{\frac{m}{k}}$$

6.4 A second example: Projectile motion with air drag

As a second example - one that is slightly more difficult while also showing off some of VPython's strengths - let us consider a projectile launched through the atmosphere. Here we don't want to neglect air drag, as is customary in introductory physics, and as we did in Chapter 5. Instead, we recall that a good formula for the drag

force (under certain assumptions) is given by,

$$\vec{F}_D = \frac{1}{2}\rho A C_D v^2(-\hat{v}), \tag{6.6}$$

where ρ is the density the medium (here, air), A the cross-sectional area of the projectile, and C_D the coefficient of drag. We see that the magnitude of the drag force is proportional to the square of the speed, and that its direction is opposite to the motion. This latter point is mathematically represented by the last term in Equation (6.6), where $\hat{v}$ is the unit vector in the direction of instantaneous motion, or $\hat{v} = \vec{v}/|v|$.

You might be thinking that the changing direction of the drag force would be difficult to "handle", and ordinarily you would be right. In fact, this facet of the problem coupled with the quadratic dependence on speed makes this problem impossible to solve in closed form with pencil and paper. So here we now encounter our first instance of a problem that has no closed-form analytical solution and where we rely on a computer to give us the solution numerically.

We said "ordinarily" because within the VPython environment there exist high-level commands that will make this problem substantially easier. Particularly nice are the built-in vector operations that include easy evaluations of *dot* and *cross*-products between vectors, as well as evaluations of a vector's magnitude and direction. For our purposes here, we want to highlight two operations we can perform on a vector **A**:

- **mag(A)** and **mag2(A)**, where the output yields the length and length-squared of that vector, respectively.
- **A.hat**, which produces from the vector **A** its unit vector (by dividing it by its length). This syntax treats the directionality (given by the unit vector) as a *property* of the vector. It is just like any other property of a vector, such as **A.x**, which yields the x-component of the vector **A**.

Armed with these commands, implementing Equation (6.6) should seem a lot more straightforward. Let's look at the basic code first (see Figure 6.2).

```
1  GlowScript 2.7 VPython
2
3  scene.center=vector(50,20,-10)
4  scene.fov=pi/2.5
5
6  box(pos=vector(50,0,0),size=vector(100,0.1,5),color=color.yellow)
7
8  ball=sphere(pos=vector(0,0,0), radius=0.2, color=color.green, make_trail=True)
9
10 speed=65
11 angle=45/180*pi
12
13 vx=speed*cos(angle)
14 vy=speed*sin(angle)
15 ball.v=vector(vx,vy,0)
16
17 m=2.0; rad=0.2; g=9.8
18 dt=0.001
19 rho=1.2; C=0.5; Area=pi*rad**2
20
21 while ball.pos.y>-0.01:
22     rate(400)
23     F=vector(0,-m*g,0) - 0.5*C*rho*Area*mag2(ball.v)*ball.v.hat
24     a=F/m
25     ball.v = ball.v + a*dt
26     ball.pos = ball.pos + ball.v*dt
27 print("The range was:", ball.pos.x, "meters.")
```

Figure 6.2: The basic code simulating a the trajectory of a projectile with air drag

One thing to point out before we delve in is that everything here is in SI-uits. This means that whenever you encounter a number (assigned to a variable), that number should be considered to have the appropriate SI-unit. So, for example, Line 10 states: **speed=10**. This means that we set the speed of the projectile to 10 m/s. Similarly, Line 12 defines the density, **rho = 1.2**, as 1.2 kg/m^3 (the density of air). Since the SI-unit system is closed, any calculations that the code performs will automatically be in the appropriate SI-unit for that variable.

We can start by looking at Lines 13 through 15. Here we first compute the x- and y- components of the velocity vector from the speed

and launch angle. Line 15 is interesting: **ball.v = vector(vx,vy,0)**. How should we interpret this statement? First, it is an assignment. The velocity vector on the right of the equal sign is assigned to a quantity called **ball.v**. The notation on the left side of the equal sign suggests that **v** is a property of **ball**. In fact, with this statement we define the velocity to be a property of the object we introduced earlier in the code and called **ball**. So now in addition to the properties **pos**, **size**, and **color** by virtue of the **ball** being defined as a sphere, we have added the property **v**.

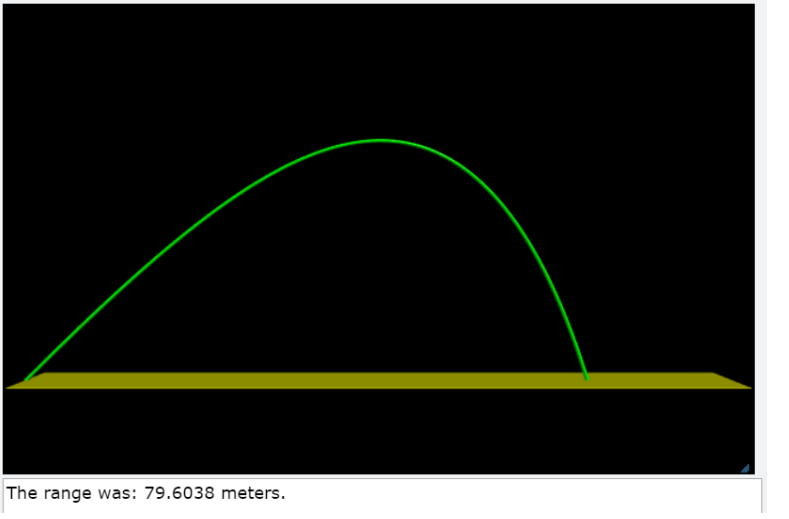
What is interesting is that **ball.v** has itself all the properties that a vector can have, and so, for instance **ball.v.x** would refer to the x-component of the ball's velocity vector. Similarly, **ball.v.hat** would give the unit vector in the direction of motion.

Finally, let's examine Line 23:

F=vector(0,-m*g,0) - 0.5*C*rho*Area*mag2(ball.v)*ball.v.hat.

You should recognize this as the net force acting on the projectile. The first part is the gravitational force near the surface of the earth and the second is the drag force. Notice that VPython automatically adds these two parts as vectors.

When you run the program you should get something like the following:



Notice that the code returns the approximate range of the projectile in the print statement. Also observe the shape of the trajectory. It clearly deviates from the parabola - the highest point in the trajectory is not reached at the horizontal half-way point, and the descent is steeper then the ascent.

6.5 Exercises

- Run the program with different launch angles and determine the angle that gives you the largest range. Without air-drag, it is fairly easy to prove that that angle is 45 degrees. What is it now?

- Let's play a game. The object is to make the range as close to 100 meters as possible. You are only allowed to change the initial launch angle and speed. What combination gets you closest to 100 meters? There may be more than one solution.

- Instead of modifying the initial conditions, as in the previous exercise, now examine the role of the projectile properties. The shape of the projectile enters the problem via the drag coefficient C_D; it can vary from as low as 0.05 (for a very aerodynamic shape) to about 1.0 (for a cube). Adjust first the mass and then the drag coefficient, and describe how these two parameters affect the trajectory.

- Add to the code given above so that you can get two trajectories on screen simultaneously. These two trajectories should correspond to two different projectiles (differentiated either by mass or drag coefficient).

6.6 Third Example: Trajectories of Planets and Comets

Our final example is also the most famous in the sense of historical significance - the Keplerian orbits of planets and comets around the sun. The perfectly circular orbit is one special solution to Newton's second law. This is usually demonstrated in introductory physics by setting the formula for the centripetal force equal to the gravitational force,

$$\frac{mv^2}{r} = \frac{GMm}{r^2}, \tag{6.7}$$

We set them equal because gravitation actually provides us with the centripetal force necessary for moving in a circle. Solving Equation (6.7) for v yields the orbital speed as a function of the orbital radius,

$$v = \sqrt{\frac{GM}{r}} \tag{6.8}$$

Proving that elliptical orbits also satisfy the governing equations is much more difficult and usually reserved for an junior-level course in classical dynamics. The same goes for parabolic and hyperbolic orbits - the remaining cone sections. However, we can explore those orbits numerically using VPython without any advanced knowledge of physics.

Imagine for a moment that we had the power to launch a comet at a particular distance from the sun, call it r_0, as well as with a certain initial velocity. It is not hard to see that if we chose a velocity of zero, the comet would head straight for the sun; it would accelerate in a straight line toward the sun and be swallowed up by it. In fact, many people unfamiliar with physics think that this scenario is the only one possible and that circular orbits are only feasible due to other planets or stars in the picture, or by some other magic (which is, of course, ludicrous).

But what if we chose an initial velocity at right angles to the line connecting the comet to the sun? We would at least have a chance of obtaining a circular orbit, but only if the speed in this direction matched Equation (6.8). Indeed, this set of initial conditions would produce a circular orbit.

Now imagine what would happen if the speed that we impart to the comet at right angles did not match that speed. What if it were much smaller or larger than what Equation (6.8) demands? It stands to reason that we would then not recover a circular orbit, but what do we get instead? Let's find out by running a numerical simulation!

The basic code is surprisingly short. In this version we decided for simplicity to set the universal gravitational constant to 1, but you can change it to the actual value in SI-units. In that case, however, you should also make the masses and distances involved realistically large. The basic code, then, is shown in Figure 6.6. Feel free to make

the parameters more realistic; the mass ratio that appears here is only 1:100, for instance. Nonetheless, we can use this simplified code to explore the possible orbits.

```
GlowScript 2.7 VPython

sphere(pos=vector(0,0,0),radius=1,color=color.yellow)
planet=sphere(pos=vector(10,0,0), radius=0.2,color=color.green, make_trail=True)
#scene.camera.follow(planet)
planet.v=vector(0,1.5,0)
m=1; G=1
M=100
dt=0.005

while True:
    rate(400)
    r=planet.pos
    F=-G*M*m*r.hat/(mag2(r))
    a=F/m
    planet.v=planet.v + a*dt
    planet.pos=planet.pos + planet.v*dt
```

Figure 6.3: The basic code simulating gravitational orbits.

6.7 Exercises

- Examine the code provided, make sure that you understand what each line does, and annotate.

- Use Equation (6.8) with the parameters given in the code to find the orbital speed for a circular orbit. Enter this speed as the starting speed in the code at the appropriate place. Do you get a circular orbit?

- If you doubled the initial distance of the comet from the sun, what speed would be necessary now for a circular orbit? Try it in the code.

- Now select a starting speed that is smaller than the one you chose above. Observe the kind of orbit you obtain now. Does the orbit trace out the same path after each revolution? Astronomers call this scenario a closed orbit.

- What can you say about the orbital speed? Is it constant or does the comet appear to speed up at certain points? Explain!

- If you make the initial speed too small, you will see that the simulation eventually breaks down and returns nonsensical results. Why is that and what can you adjust to remedy the situation? [Hint: When the comet gets very close to the sun, the acceleration it experiences becomes very large. What line in the code is going to be adversely affected?]

- Let's be even more creative. Instead of the normal law of universal gravitation that decreases with the square of distance, what would happen in a universe ruled by a gravitational force proportional to $1/r$? Make this change in the code, and start with a circular (or near circular) orbit by using Equation (6.7) but with the new gravity to solve for orbital speed.

- Now lower the speed and observe the orbits that result. What is the most obvious qualitative difference about the orbits that we get now in this alternative universe?

Conservation of Momentum and Energy

7.1 A binary-star system

In the previous chapter we considered the motion of a planet subject to a central force that was always directed from the planet towards the origin where a star was sitting motionless. This is, of course, not quite correct, since the star also experiences a gravitational pull towards the planet and must therefore also move. This motion is usually very small since stars tend to be much heavier than planets, but it is not zero.

Let's consider now the full two-body problem. To give us maximal flexibility in parameters, we can consider the two bodies both stars in a binary-star system. We will creatively call the heavier star "bigstar" and the lighter star "smallstar". We could, of course, proceed exactly like before, modifying the code in Chapter 6 to include the dynamics of the second object (i.e., the sun). But to illustrate another important physical concept, namely that of momentum and momentum conservation, let us incorporate this quantity in the code.

To review briefly, momentum is defined by $\vec{p} = m\vec{v}$. An object has momentum by virtue of having mass and velocity. We can also talk of the total momentum of a system of objects. In that case we simply add the individual momenta of all the objects that make up the system together (as vectors). Now, we can prove that when this system is *isolated*, in other words when nothing outside this system exists that would push or pull on the objects within the system, then the total momentum of the system doesn't change. We say that the total momentum is *conserved*.

Another useful concept in this context is the *center of mass* of a system comprised of discrete objects/particles. We can compute the coordinates of the center of mass using the well-known formula,

$$x_{com} = \frac{m_1 x_1 + m_2 x_2 + ...}{m_1 + m_2 + ...},$$

and similarly for the y-coordinate. Since VPython deals well with vector quantities, we can also dispense with the individual components and refer only the position vectors, $\vec{r}_i$, of the objects themselves:

$$\vec{r}_{com} = \frac{\sum m_i \vec{r}_i}{\sum m_i}. \tag{7.1}$$

You may have learned that whenever the total momentum of a system is conserved, i.e., when there is no net external force acting on the system, then the center of mass cannot experience any acceleration. In our notion, we can write,

$$\vec{a}_{com} = \frac{d^2}{dt^2}(\vec{r}_{com}) = 0. \tag{7.2}$$

The proof of this statement is actually not difficult and well within the reach of an introductory physics student. See if you can work it out yourself.

One of our objectives will now be to "prove" this property about the center of mass numerically by simply computing its location for each time-step and then to observe its motion across the screen. Another will be to show that in this binary-star system, total momentum will be conserved. It should be conserved because the way we have set up the problem, the system is clearly isolated - there are no other objects around at all.

```
GlowScript 2.7 VPython

scene.range=5e11

G = 6.7e-11

bigstar = sphere(pos=vector(-2e11,0,0), radius=2e10, color=color.red,
                 make_trail=False, interval=10)
bigstar.mass = 3e30
bigstar.p = vector(0, 3e3, 0) * bigstar.mass

smallstar = sphere(pos=vector(2e11,0,0), radius=1e10, color=color.yellow,
                 make_trail=False, interval=10)
smallstar.mass = 1e30
smallstar.p = vector(0, -1e4, 0) * smallstar.mass

dt = 1e5

centerofmass =
(bigstar.mass*bigstar.pos+smallstar.mass*smallstar.pos)/(bigstar.mass+smallstar.mass)

COM = cone(pos=centerofmass, axis=vector(0,1e10,0),radius=1e10,color=color.green,
make_trail=True)

while True:
    rate(200)
    r = bigstar.pos - smallstar.pos
    F = G * bigstar.mass * smallstar.mass * r.hat / mag2(r)

    bigstar.p = bigstar.p - F*dt
    smallstar.p = smallstar.p + F*dt

    bigstar.pos = bigstar.pos + (bigstar.p/bigstar.mass) * dt
    smallstar.pos = smallstar.pos + (smallstar.p/smallstar.mass) * dt
    COM.pos = (bigstar.mass*bigstar.pos+smallstar.mass*smallstar.pos)/(bigstar.mass+
    smallstar.mass)

    print ("The total momentum is", bigstar.p + smallstar.p)
```

Figure 7.1: The code for the binary-star system.

Let us again start with a code template - see Figure 7.1. One thing you will notice immediately is that we are now not working in artificial units (as before) and instead use the true value for the gravitational constant. That also forces us to use astronomically realistic values for all other quantities in the problem, such as for the masses and distances involved. Again, as long as all the inputs are in SI-units, so will all the computed outputs.

As you can see, in the first six command lines, we basically set up the properties about the two stars. For instance, the bigger star has a mass of 3×10^{30} kg - three times that of the smaller star. For

comparison, our sun's mass is about 2×10^{30} kg. One of the things you will have a chance to play around with is the mass ratio of the two stars.

The next line (Line 17) defines the computational time step. This might strike you as an incredibly large time step, especially compared to the values we have used before, but remember - everything has to be astronomical, including time. The time step of 10^5 seconds translates to a duration of slightly longer than an earth day. Compared to the period of revolution this is still quite small.

Lines 19 and 20 are recognized from Equation (7.1) as nothing other than $\vec{r}_{com}$. The next two lines are there just so we can visualize the location of center of mass on the screen, here in the aspect of a green *cone* whose trajectory we will also keep track of via the "make_trail" command.

The iterative part of the program starts with Line 25. We define the vector, **r**, that runs from the small star to the big star. We could have also reversed it, but it is important to be consistent. The way it is defined, it will be parallel to the force on the small star and anti-parallel to the force on the big star. By Newton's third law, these two interaction forces must be equal and opposite.

Lines 30 and 31 implement Newton's second law,

$$\vec{F} = \frac{d\vec{p}}{dt}. \tag{7.3}$$

If we solve this for the small change in momentum, we get: $dp = Fdt$; in other words, it takes a force to change the momentum. This is also sometimes referred to as the *impulse-momentum* theorem. Once we have calculated the new momentum, we can also update the position of the corresponding star, since momentum, of course, is intimately related to velocity - we get velocity by dividing the momentum by mass. This way, we do not have to explicitly refer to

velocity at all in the WHILE loop.

7.2 Exercises

- Run the program - does the computed total momentum change over time? Is physics correct? Also examine the code - is the result at all surprising?

- You will also notice that the center of mass does not stay stationary, but that it moves on the screen. We can make that motion cease if we give the two stars initial momenta that add to zero. Show that right now (i.e., in Figure 7.1) they two initial momenta do not add to zero.

- Now change the initial conditions such that the total momentum is in fact zero initially. What do you notice about the center of mass motion on the screen? Also change the viewer's perspective to see the two-body motion from different angles.

- Make an additional change in initial conditions (positions and velocities) and watch the orbits of the two stars around each other. Consider turning on the "trails" on the orbits for better visualization.

- VPython allows you to view the motion from the perspective of different observers (or reference frames). The command is **scene.camera.follow(bigstar)**. In the parenthesis appears the name of the object that you want to make as the reference frame. In the command above we take the big star as the observer's reference frame. To make things less confusing,

turn off all the trails again.

- Now that you have explored the role of initial conditions a bit, let's turn to the mass ratio. Try one case where the ratio is much larger than 3, and then one case for the ratio is close to 1. What do you conclude from those two scenarios.

7.3 Energy in the Spring-Mass System

In Section 6.2, we explored the motion of a mass at the end of a spring. We saw the sinusoidal motion that resulted from the combination of Newton's second law and Hooke's law. Now we can revisit this problem and analyze it through the lens of potential and kinetic energy.

Remember that the potential energy stored in a stretched or compressed spring, also called elastic potential energy, is given by $U = \frac{1}{2}kx^2$. Furthermore, the kinetic energy of the end-mass is given by $K = \frac{1}{2}mv^2$. Finally, the total mechanical energy is simply the sum of the two energies, $E_{mech} = U + K$.

Modify the code, as given in Figure 6.1, in the following manner.

- Add some lines that will compute the three energies, U, K, and E_{mech}.

- Then, instead of plotting position and velocity as a function of time, instead plot the three energies that are now being computed at each time step. What do you observe? Is the total mechanical energy conserved over time? Explain!

7.4 Simulating the Rutherford experiment

Another example that we will see is actually mathematically very similar to the binary-star system is the famous Rutherford scattering experiment. We know that two like charges repel via the Coulomb force. This force has the same structure as the law of universal gravitation. It is given by,

$$\vec{F}_c = k\frac{q_1 q_2}{r^2}\hat{r}, \tag{7.4}$$

where the Coulomb constant $k = 9.0 \times 10^9$ N m^2/C^2 plays the role of the gravitational constant, G. Notice how this force also depends inversely on the square of the distance between the two objects, and that it is again a *central* force (indicated by the direction vector $\hat{r}$).

Ernest Rutherford's experiment was to shoot alpha particles, which are fast Helium nuclei, at a thin gold foil. The idea was to see if and how the alpha particles would be deflected upon hitting the foil. What Rutherford discovered was that every once in a while the deflection angle was very large so that the alpha particle was scattered back to the source. The explanation for these large deflections is that the positive charge in the gold atoms is actually highly concentrated in the nucleus (and not smeared out). In fact, while the size of the gold atom is on the order of 10^{-10} meters, the nucleus is 100,000 times smaller, i.e., on the order of 10^{-15} meters.

So if the positively charged alpha particle hits a gold nucleus close to head-on, since the two repel it should be deflected back. At the same time, the Gold nucleus should also experiences some recoil. (If it didn't, momentum could not be conserved during the collision.) Let us now use VPython to explore the different scenarios that can

occur when we shoot an alpha particle at a gold nucleus.

While Equation (7.4) is very similar in form to Equation (6.7), the length scales of the nuclear-collision problem could not be more different from that of the binary-star problem. In the nuclear problem, a typical distance (or *characteristic* length scale) is on the order of 10^{-15} m, whereas in the gravitational context it was 10^{+11} m - an incredible difference of 26 orders of magnitude. With this change in spatial scale come other difference, such as in characteristic mass and time. In short, we are now dealing with vastly smaller lengths, vastly tinier masses, and vastly shorter times. Luckily for us, VPython can handle all of these changes very well.

Figure 7.2 shows the new setup for the nuclear context.[1] The first thing we notice is the new **scene.range** which catapults us into the microcosm. Next we define the **nucleus** and its properties of mass, charge and momentum.

In Line 13, we introduce the so-called *impact parameter*, b. This will be an important control parameter for us to vary from run to run. This parameter tells us, roughly speaking, by how much the alpha-particle would miss the nucleus if it felt no repulsive force. Thus, $b = 0$ corresponds to a perfectly head-on collision.

Next, we define the alpha particle and its properties of mass, charge and momentum. Notice that we give the alpha particle some initial speed to the right. In fact, this initial speed is quite large, $v_0 = 5 \times 10^7$ m/s, about 17 percent of the speed of light. We need large speeds for the alpha particle to get close to the gold nucleus. This initial speed is another parameter we will be able to adjust later.

Line 22-25 set up arrows for the momenta of the two particles (alpha

[1] This code is loosely based on the code by R. Chabay, "03-particle collision", available in the Matter-and-Interactions's Glowscript-programs folder.

```
1  GlowScript 2.8 VPython
2
3  scene.range = 3.5e-14
4
5  k = 9e9
6
7  nucleus = sphere(pos=vector(0,-1e-14,0), radius=2e-15, color=color.red, make_trail=True)
8  nucleus.mass = 1.32e-25
9  nucleus.q = 79*1.6e-19
10 nucleus.p = vector(0, 0, 0)
11
12 #The impact parameter, b
13 b = 1.5e-15
14
15 alpha = sphere(pos=vector(-2.5e-14,b-1e-14,0), radius=1e-15, color=color.yellow, make_trail=True)
16 alpha.mass = 6.64e-27
17 alpha.q = 4*1.6e-19
18 alpha.p = vector(5e7, 0, 0) * alpha.mass
19
20 dt = 5e-24
21
22 scale=2.5e4
23 p1=arrow(pos=vector(1e-14,1e-14,0),axis=alpha.p*scale, color=color.yellow, shaftwidth=7e-16)
24 p2=arrow(pos=vector(1e-14,1e-14,0),axis=nucleus.p*scale, color=color.red, shaftwidth=7e-16)
25 ptot=arrow(pos=vector(1e-14,1e-14,0),axis=(nucleus.p+alpha.p)*scale, color=color.white, shaftwidth=7e-16
26
27 #Start-Pause Button
28 run = False
29 def Runbutton(r):
30     global run
31     run = not run
32     if run:
33         r.text = "Pause"
34     else:
35         r.text = "Run"
36 button(text='Run', bind=Runbutton)
```

Figure 7.2: Setting up the Rutherford-scattering code

and Gold nucleus) and the total momentum. In order to see these arrows on the screen, they have to be stretched by the scale-factor, **scale**.

Finally, we have added here for the first time a start-pause button, which will allow us to start the code and pause it at any time. The details of this section of code need not concern us here.

The main part of the code, namely the WHILE loop, is shown in Figure 7.3. We should recognize large chunks of it from before. Lines 52 to 55 are new. They will update the momentum vectors throughout the interaction of the two particles. The lengths and direction of the three momenta (alpha, nucleus, and total) are set via the **.axis** assignment. Additionally, we have displaced the *tail* of the momentum vector associated with the nucleus to coincide with the *head* of

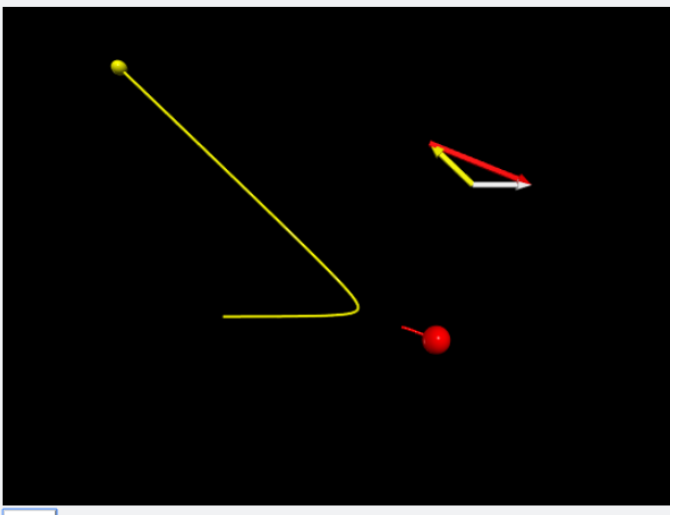
the alpha's momentum vector. This way we can verify visually that the two individual momentum vectors do indeed add to the same total momentum vector (via the head-to tail method of adding vectors).

```
while True:
    rate(50)
    if not run: continue

    r = alpha.pos - nucleus.pos
    F = k * alpha.q * nucleus.q * r.hat / mag2(r)

    alpha.p = alpha.p + F*dt
    nucleus.p = nucleus.p - F*dt

    nucleus.pos = nucleus.pos + (nucleus.p/nucleus.mass) * dt
    alpha.pos = alpha.pos + (alpha.p/alpha.mass) * dt

    p1.axis = alpha.p*scale
    p2.axis = nucleus.p*scale
    p2.pos = vector(1e-14,1e-14,0) + alpha.p*scale
    ptot.axis = (alpha.p + nucleus.p)*scale
```

Figure 7.3: The core of the Rutherford-scattering code

7.5 Exercises

- Run the code and observe the trajectories of the two particles. Also observe the particle's momentum vectors in the upper-right corner. You should see something like this:



- Verify that the momentum vectors are in fact tangentially parallel to the trajectory at any given instant of time. Why is the red arrow so large if the gold nucleus seems to move so slowly?

- Given the vector diagrams rendered in the upper-right corner of the screen, is momentum conserved throughout the scattering process? How do you know?

- Adjust the impact parameter, b. Make it progressively smaller (all the way to zero) and observe what happens to the deflection angle of the alpha particle.

- Now make b incrementally bigger than the value in Figure 7.2. What do you see now? Is momentum still conserved?

- You can now explore the role of the initial speed of the alpha particle. How does the angle of deflection seem to depend on this initial speed?

Rotational Motion, Torque, and Angular Momentum

In the previous chapters you saw how a binary star system or a comet can be simulated in VPython. In this chapter, we will review the motion of a comet, and explore torque and angular momentum. The physics of a binary star system and a Sun-comet system are exactly alike. As a matter of fact, the same physics governs any gravitational interaction (in a classical mechanics sense): a cluster of stars, a moon and its planet, or a cluster of galaxies.

8.1 The Physics

Recall Newton's Law of Universal Gravitation:

$$\vec{F} = -\frac{GMm}{r^2}\hat{r} \tag{8.1}$$

(Note this looks similar to Equation (7.4), the Coulomb's Law force between two charges).

Note that the direction of the position unit vector $\hat{r}$ is always pointing in the exact opposite direction (because of the minus sign) of the force vector.

Angular momentum (symbol $\vec{L}$), is the physical quality that is the rotational equivalent of linear momentum. Remember that linear momentum $\vec{p} = m\vec{v}$. Angular momentum $\vec{L} = I\vec{\omega}$, where I is the *moment of inertia*, and $\vec{\omega}$ we encountered way back in Chapter 3 - it is called the angular velocity. **Torque** (symbol $\vec{\tau}$) can be thought of as the rotational equivalent of force. It is determined from the cross

product $\vec{r}$ and $\vec{F}$

$$\vec{\tau} = \vec{r} \times \vec{F} \tag{8.2}$$

Equation (8.2) shows that the torque is the **cross product** of the position vector $\vec{r}$ and the force applied $\vec{F}$. The cross product is a vector. Remember that the magnitude of the cross product here is

$$|\vec{r}||\vec{F}| \sin\theta,$$

where θ is the angle between $\vec{r}$ and $\vec{F}$. Its direction is perpendicular to both of the two vectors in the cross product. Here, therefore, the torque $\vec{\tau}$ is perpendicular to both $\vec{r}$ and $\vec{F}$.

It is very easy to program the cross product in VPython. To calculate the cross product of $\vec{A}$ and $\vec{B}$, one uses the command

```
cross(A,B)
```

Remember that A and B **must** have been declared to be vectors in VPython. VPython will know automatically that the value assigned to `cross(A,B)` is a vector implicitly.

Just as force is the change of linear momentum with respect to time, see Equation (7.3), torque is the change of angular momentum with respect to time:

$$\vec{\tau} = \frac{d\vec{L}}{dt} \tag{8.3}$$

Furthermore, just as linear momentum is conserved in the absence of an outside force, angular momentum is conserved in the absence of an outside torque. Note that this follows from Equation (8.3).

Look back at Equation (8.1). Recalling that the unit vector $\hat{r}$ is rotated 180° from $\vec{F}$, what is the value of $\vec{\tau}$?

```
GlowScript 2.8 VPython

scene.forward = vector(0,-.3,-1)
scene.height=600
scene.width=1200
scene.range=3e11

G = 6.7e-11 # Newton gravitational constant

sun = sphere(pos=vector(0,0,0), radius=2e10, color=color.yellow,
             make_trail=True, trail_type='points', interval=10)
sun.mass = 2e30

comet = sphere(pos=vector(-1.5e11,0,0), radius=1e10, color=color.white,
             make_trail=True, interval=10)
comet.mass = 1.25e13
comet.p = vector(0,-41.89e3,0)*comet.mass

sun.p = -comet.p
dt = 1e5
t=0
while True:
    rate(2000)
    r = comet.pos - sun.pos
    F = G * sun.mass * comet.mass * r.hat / mag2(r)

    sun.p = sun.p + F*dt
    comet.p = comet.p - F*dt
    sun.pos = sun.pos + (sun.p/sun.mass) * dt
    comet.pos = comet.pos + (comet.p/comet.mass) * dt
    t=t+dt
```

Figure 8.1: Program for Comet Orbiting the Sun

Because $\tau = 0$, then $\vec{L}$ must be conserved.

See the sample program and output in Figure 8.1.

8.2 Exercises

- Add to the code in Figure 8.1 to calculate the torque and plot it (remember it is a vector and you can print the components of a vector!).

Is it pointing in the correct direction?
Show that the program is calculating a torque that is close to the correct answer (it may not, due to limitations in how precisely VPython does calculations).

- Write code to calculate the angular momentum and plot it. Is *it* pointing in the correct direction? Print the angular momentum with every iteration. What do you notice? Does it make sense?

- Why is the Sun not appearing to move?

- Change the parameters of the program to illustrate another scenario (for example, the Earth revolving around the Sun). Discuss what you see.

Two Capstone Projects

9.1 The Gravitational "Slingshot"

In the previous chapters of this book you have gone from investigating the simple motions of an object with constant velocity to exploring the physics of a comet about the Sun. While these phenomena are "real life", we want to end with a consideration and discussion of a truly "Space Age" phenomenon that is used regularly in sending space probes to the outer solar system – **gravitational assist**, or the "slingshot" maneuver.

The outer planets – Jupiter, Saturn, Uranus, and Neptune – as well as the objects in the Kuiper Belt – including Pluto, Makemake, and $2014\ \mathrm{MU}_{69}$ – are extremely far away. Jupiter is just over 5 astronomical units[1] away from the Sun, Neptune is 30 a.u. away from the Sun, and Pluto is 40 a.u. away from the Sun. Traveling to any of these objects will take several years to get there.

But *how* to get there? Traveling directly away from the Sun to arrive at a distant planet would take an extraordinary amount of fuel to travel – and that extra fuel means more mass, and the more mass, the more fuel needed. It would be extremely costly, and at this writing we do not have the technology to develop rockets to transport a space probe those distances. The Hohmann orbit (least energy method) saves on fuel, but takes quite a while to complete the journey.

Gravitational assist allows a space probe to take some of the energy from a nearby planet and use it for itself. The energy is "free"!

[1] An astronomical unit is the average distance of the Earth to the Sun. It is equal to 1.50×10^{8} kilometers.

9.1.1 The Physics

The slingshot maneuver involves a planet and a space probe. Both objects have a velocity (therefore, both a speed and a direction). When the probe nears the planet and is appreciably affected by the planet's gravity, it accelerates (both in speed and direction). The interaction is a "collision", but generally nothing collides. Furthermore, the collision is elastic, so therefore there is no loss of energy (and, of course, no change of total momentum) in the planet-probe system.

So, if the kinetic energy of the probe increases, then the kinetic energy of the planet *d*ecreases. The planet slows down! However, the amount of decrease of kinetic energy is negligible to the planet, because the mass of a planet is **huge** compared to the mass of a space probe. For example, the mass of Uranus or Neptune is approximately 10^{26} kg, compared to the mass of a space probe, which is typically on the order of 10^3 kg.

In the rest frame of the Sun, the Sun "sees" the planet moving with initial velocity $\vec{U}_i$. The space probe is moving with initial velocity $\vec{v_i}$. The two objects are not necessarily going to meet head-on; that would certainly result in the loss of the spacecraft. Instead, their velocity vectors are separated by a distance b, called the **impact parameter**. If $\mathrm{b} \leq \text{the radius of the planet } \mathrm{R_p}$, then there *would* be a head-on collision.

Consider a planet and a space probe. In the Sun's frame of reference, the probe has velocity $\vec{v}$ and the planet has velocity $\vec{U}$ (see Figure 9.1). Therefore the probe's kinetic energy is $\frac{1}{2}mv^2$. In the planet's frame of reference, the planet sees the probe approach with speed $v + U$. As it gets closer, the speed of the probe increases, and after the encounter, the probe's speed decreases until it appears to the planet to have a speed $v + U$ again, as shown in Figure 9.2. (We are

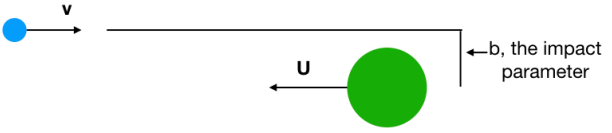


Figure 9.1: Space Probe Approaching Planet

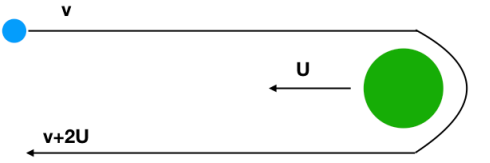


Figure 9.2: Space Probe After Experiencing Gravitational Assist From Planet

assuming an ideal encounter that causes the probe to make a 180° change of direction). Therefore, in the Sun's frame of reference, it appears as if the probe has speed $v + 2U$.

In the Sun's frame of reference, then, the kinetic energy of the probe is $\frac{1}{2}m(v + 2U)^2$. The increase of energy to the probe is $2mU(v+U)$. (By the same token, because the energy is conserved in this elastic collision, the planet's kinetic energy goes down by the same amount, but the change is negligible for the planet).

Several probes to the outer Solar System have used gravitational assist. They include the *Cassini* mission to Saturn (see Figure 9.3), the *New Horizons* mission to Pluto and 2014 MU_{69}, and the famous

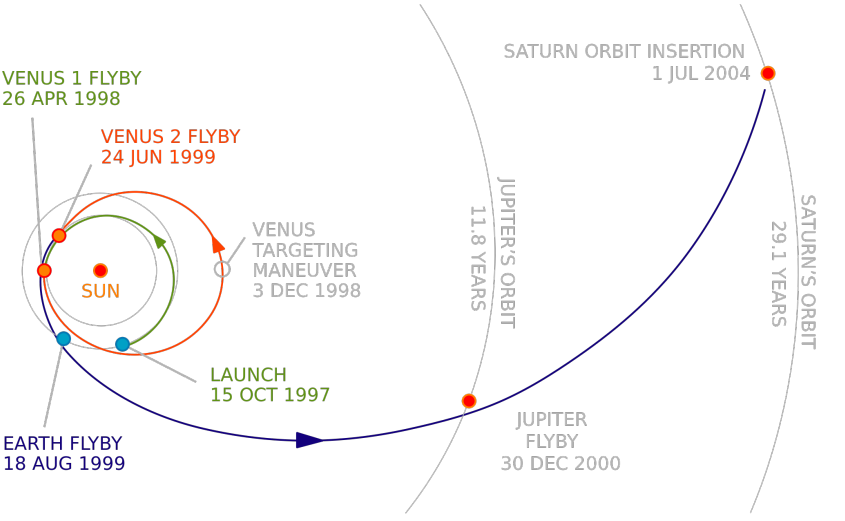


Figure 9.3: Trajectory of the *Cassini* Mission to Saturn (source: NASA)

probes *Voyager* 1 and 2, which are now in interstellar space.[2]

9.1.2 The Code

In Figure 9.4 the kinetic energies of the probe and the planet are calculated (e.g., in line 11) as $\frac{p^2}{2m}$, which is the same as $\frac{1}{2}mv^2$. In line 27, the gravitational force is calculated, and lines 34 and 35 update the planet's and the probe's positions at each instant.

[2] *Voyager* 1, launched in 1977, is now almost 150 a.u. from the Sun and is the farthest human-made object in existence.

```
GlowScript 2.8 VPython

scene.range=2e9

G = 6.7e-11

planet = sphere(pos=vector(2e8,4e8,0), radius=5e7, color=color.red,
                make_trail=True, interval=10)
planet.mass = 1e26
planet.p = vector(-1.5e3, 0, 0) * planet.mass
planet.K = mag2(planet.p)/(2*planet.mass)

probe = sphere(pos=vector(-2.0e9,5.2e8,0), radius=1e7, color=color.yellow,
                make_trail=True, interval=10)
probe.mass = 1e4
probe.p = vector(5e3, 0, 0) * probe.mass
probe.K = mag2(probe.p)/(2*probe.mass)
print("Initial KE of Probe:", probe.K, ".  Initial KE of Planet", planet.K)

dt = 2e2
r = planet.pos - probe.pos
r0=r

while mag(r)<=mag(r0):
    rate(300)
    r = planet.pos - probe.pos
    F = G * planet.mass * probe.mass * r.hat / mag2(r)

    planet.p = planet.p - F*dt
    probe.p = probe.p + F*dt
    probe.K = mag2(probe.p)/(2*probe.mass)
    #print(satellite.K)

    planet.pos = planet.pos + (planet.p/planet.mass) * dt
    probe.pos = probe.pos + (probe.p/probe.mass) * dt
print("Final KE of Probe:", probe.K, ".  Final KE of Planet", planet.K)
```

Figure 9.4: Sample Code for Gravitational Assist

9.1.3 Exercises

1.
   - Run this program to make sure it works. Describe what you see.

   - Modify the code to incorporate an impact parameter, b.
   - What happens when you change the impact parameter, to make it larger or smaller?

   - What happens when you change the velocities of the probe or the planet? In particular, what happens to the speed of the probe when the planet's velocity is 0?

   - Similarly, what happens when the planet's velocity changes sign? In that scenario, the probe and the planet are initially moving in the same general direction, with the probe catching up to the planet.

2.
   - Now that you have explored the b-dependence a bit, let's make this a bit more precise. Choose five different impact parameters and keep track of the angle of deflection of the probe for each. Enter the values for b and deflection angle into a spreadsheet and plot the relationship between those variables. If the graph is not clear, go back and add a few more b-values into your table.

   - For the same b-values that appear in your list, write down the the change in kinetic energy of the probe, expressed as a fraction of the initial kinetic energy, based on the code output. Then enter these values for fractional change in kinetic energy into your spreadsheet and plot its dependence on impact parameter.

Based on this plot, if we want a large increase in kinetic energy for the probe, what do we have to do?

9.2 Asteroid near a Binary Star System - Chaotic orbits, or the "Three-Body Problem"

In the previous chapters you have gone from investigating the simple motions of an object with constant velocity to exploring the physics of a comet about the Sun. In all of these examples, the motion did not exhibit what physicists call *deterministic chaos*. In the gravitational problem of two bodies attracting one another, solutions can, in fact, be found by pencil and paper. Interestingly, when we add a third body all bets are off. The orbits can no longer be predicted with a mathematical formula, but worse than that, the orbits exhibit what is known as *extreme sensitivity to initial conditions* - a hallmark of chaos.

9.2.1 Modifying a previous code

To keep things as simple as possible, let us revisit an earlier problem - the binary-star system in Section 7.1. The question we want to ask now is: if we introduced a small asteroid into this system how would it move under the influence of the gravitational attraction to each star? To make the problem even simpler, we assume that the asteroid is so small that it does not affect the motion of the two stars. Since it is 8 orders of magnitude (or 100 million times) less massive than the stars, this is a very good approximation. In that case, the mass of the asteroid really doesn't enter into the picture. All we need to do is to calculate the acceleration experienced by the asteroid at

every time step, and this is given by the universal law of gravitation as,

$$\vec{a} = -G\frac{m_1}{r_1^2}\hat{r}_1 - G\frac{m_2}{r_2^2}\hat{r}_2, \tag{9.1}$$

where $\vec{r}_1$ is the vector from first star to the asteroid, and $\vec{r}_2$ is the vector from the second star to the asteroid, and the m's in the numerator are the star masses.

Once we have computed the accelerations, we can update the asteroid velocity, and then we use the updated velocity to find the next position of the asteroid, in a recursive procedure outlined in Chapter 6.

So here is the idea. Let us start with the code in Figure 7.1 which will take care of the motion of the two stars. To give us a little more "room", let's start these two stars out along the x-axis at -5e11 and +5e11 (instead of $\pm$2e11. Also, to keep the center of mass of the system fixed, let's adjust the y-component of the small start from -1e4 to -9e3. Finally, turn off the trails on these two stars, and delete all calculations of the center of mass (we won't need that anymore). This takes care of the binary-star system.

Now let's introduce the asteroid. Add a line defining this object near the top that defines the asteroid as a sphere. Give it an initial position of `vector(1.50e12,0,0)`. Then define a asteroid velocity as one of its attribute as
`asteroid.v = vector(0,-7.5e3,0)`.

Now that we have defined the asteroid and its initial position and velocity, we need to turn our attention to its motion. At the end of the while-loop, add two lines that compute the vectors $\vec{r}_1$ and $\vec{r}_2$, appearing in Equation 9.1. Each of them are just the differences in the position vectors of the asteroid and the respective star. With these two vectors in hand, we can compute the asteroid's acceleration, `asteroid.a`, using Equation 9.1 directly.

```
if (mag(r1)<2e9 or mag(r2)<2e9):
    print("collision")
    break
elif (mag(r1)<8e9 or mag(r2)<8e9):
    dt = 4e3
    print("close call")
elif (mag(r1)<8e10 or mag(r2)<8e10):
    dt = 2e4
else: dt = 1e5
```

Figure 9.5: A rudimentary adaptive time step

This program would now run, but it is a good idea to add two more things. The first one is straightforward. If the asteroid comes too close it burns up in the stellar atmosphere. So we should include an if-statement to terminate the program if the asteroid comes within, say, `2e9` (or 2 million kilometers) of either star.

The second refinement has to do with the computational integration technique itself. Right now we are using a simple Euler-Cromer algorithm, as explained before. But you may have noticed that when the asteroid gets really close to either star, the dynamics is no longer smooth. The asteroid picks up so much speed that it covers a significant amount of space between successive time steps. In other words, the computational time interval, dt, seems to be too large, and the result is no longer accurate.

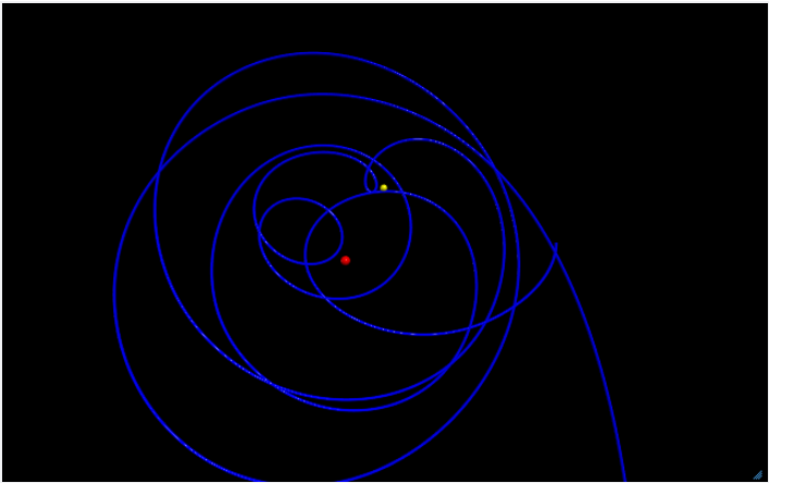
In a more advanced course, you would now probably be learning about the Runge-Kutta algorithm with an *adaptive time-step*. But that would go beyond the scope of this course. So, instead, let's implement a most rudimentary version of an adaptive time-step. The basic idea is that when the asteroid comes within a certain radius of either star, then we need to compute more finely by reducing the dt.

Figure 9.5 shows one simple way we can accomplish this. Keep in

mind, however, that this is not a highly accurate algorithm either, and that there are much more sophisticated ways to do this out there. Nonetheless, it improves on our previous method.

9.2.2 Exercises

Now that you have completed the coding, let us have fun and run the program with a few initial conditions. You should get output like the following:



One thing that is immediately clear is that the orbits are highly irregular and non-periodic. Furthermore, what you can do is to vary the initial position of the asteroid slightly. For instance, you can check the orbits that result from a initial x-coordinate of 1.49, 1.50, and 1.51×10^{12} meters.

Find initial conditions that lead to the three possible long-term outcomes for the asteroid: (a) it crashes into one of the stars, (b) it

escapes the binary-star system and drifts further and further into space, and (c) it settles into a semi-stable orbit around one of the stars.

Conclusion

We hope that the previous chapters helped you learn the material in your introductory physics textbook a little better by giving you the opportunity to explore and play with the ideas in a concrete way. We also hope that these exercises made you appreciate the power of scientific computation. It can be an invaluable tool in finding and visualizing solutions and also in helping us build up our physical intuition and our conceptual understanding of physics.

We encourage you to use the sample codes provided in this book as a springboard for developing your own codes. These programs are pretty barebones and could be embellished in numerous ways. More importantly, we hope that you use the tools you have acquired to simulate your own favorite physical processes. Play around with the code! Tinker with it, and see what happens!

Keep VPython in mind also for the second semester of your introductory physics sequence. There should be plenty of chances to use it for, say, visualizing electric and magnetic fields and so on. In fact, there are a number of excellent sample programs already on the *glowscript.org* platform. Additional programs and resources can be found at multiple online repositories.[1]

Finally, if you think you want to pursue the physics major, remember VPython can come in handy for your upper-level physics coursework. At that time, it is likely that you have become more familiar with scientific computing and may be comfortable using the Python programming language. You can choose from a number of excellent

[1] see, for instance:
https://vpython.org/contents/contributed.html
https://bphilhour.trinket.io
https://phys221.wordpress.com/vpython-tutorials/

recent textbooks on computational physics using Python.[2] In fact, we consider VPython an excellent gateway into the broader world of computational physics and scientific computing.

[2]For instance:
R.H. Landau and M.J. Páez, *Computational Problems for Physics: With Guided Solutions Using Python* (CRC Press, 2018).
J.M. Kinder and P. Nelson, *A Student's Guide to Python for Physical Modeling* (Princeton U. Press, 2018).
M. Newman, *Computational Physics* (2013).

Bibliography

R. Chabay, B. Sherwood, *Matter and Interactions* (John Wiley & Sons, 2015).

R. Chabay, B. Sherwood, "Computational Physics in the introductory calculus-based course," *Am. J. Phys.* **76**, 307 (2008).

A. Cromer, "Stable solutions using the Euler Approximation," *American Journal of Physics*, **49**, 455 (1981).

J.M. Kinder and P. Nelson, *A Student's Guide to Python for Physical Modeling* (Princeton U. Press, 2018).

R.H. Landau and M.J. Páez, *Computational Problems for Physics: With Guided Solutions Using Python* (CRC Press, 2018).

P. Laws, *Workshop Physics Activity Guide, Mechanics, Module 1 and 2* (Wiley, 2004).

L.C. McDermott, "A View from Physics" in *Toward a Scientific Practice of Science Education* (Lawrence Erlbaum Assoc., 1990).

M. Newman, *Computational Physics* (2013).

E.F. Redish and J.M. Wilson, "Student programming in the introductory physics course," *Am. J. Phys.* **61**, 222 (1993).

D. Schroeder, *Physics Simulations in Python* (2018).

D. Sokoloff, R. Thornton, P. Laws, *RealTime Physics - Active Learning*

Laboratories (John Wiley & Sons, 2012).

A. Van Heuvelen, "Learning to think like a Physicist: A review of research-based instructional strategies," *Am. J. Phys.* **59**, 891 (1991).

Authors

Windsor A. Morgan, Jr. is an Associate Professor of Physics & Astronomy at Dickinson College. He specializes in astronomy education research, and has also done research in X-ray-emitting active galactic nuclei, X-ray binary star systems, new statistical methods of studying astronomical surveys, and the formation of hydrocarbons in the early solar system.

Lars Q. English is a Professor of Physics at Dickinson College. His research explores "the nonlinear dynamics of complex systems" where both nonlinearity and lattice/network geometry play important roles in enabling and guiding processes of spontaneous patterns formation. Other interests include the Calculus of Variations, magnetism and spin resonance, microwave spectroscopy, medical imagining techniques, and issues within the philosophy of science. Visit: http://www.larsenglish.com