Introduction to Computational Physics

Physics 265

David Roundy

Spring 2011

Contents

Forward

Expectations of this course

This course straddles three subjects: Physics, Computer Science and Mathematics. In ten weeks, we won't be able to thoroughly cover any one of these. Instead, I will focus on giving you a taste of each of them, and a picture of how you can use math and computers together to deepen your understanding of Physics.

This course will focus its Physics content on Newtonian mechanics. You will also study the same Physics in other courses, but my hope is that you will understand it more deeply through this course. At the same time, you should learn some elementary programming, and will be introduced to some concepts in differential equations that you most likely will not encounter in your math classes until next year.

There is no way to learn programming, except by programming, and that is what you will be doing in this course.

Why python was chosen

We have chosen to teach this course in the Python programming language for several reasons. One is that it is particularly easy to learn, and has a wide array of online tutorials and introduction. It has a clean syntax that makes most programs easy to read. On top of all this, it is a language that is actually *used* in scientific computing, as well as in the wider programming community.

Approach used in math

We hope in this course to teach the *meaning* of calculus, not to preset proofs or carefully guarded statements. Numerical methods lend themselves to attaining an intuitive understanding of the significance of vectors, derivatives and integrals, without getting bogged down in analytical approaches that may seem obscure or even useless when they are first encountered. I hope that having seen how useful these concepts actually are, students will be eager to learn the analytical approaches that can so often lead to even deeper insights.

What is expected of classwork

- Every definition of a constant value should include units.

```
>>> ball.mass = 10 # kg
>>> ball.pos = vector(10,0,0) # meters
```

- The units should be correct in every equation in the code.
- The argument to sin or cos must be unitless, and that should be clear from the code.

```
>>> x = sin(t) # bad
>>> omega = 1 # 1/s
>>> x = sin(omega*t) # good
```

- Use vector operations when possible.

```
>>> dr = r1 - r2 # good
>>> dr = vector(r1.x-r2.x, r1.y-r2.y, r1.z-r2.z) # bad
```

- When possible, complicated equations should be broken down into separate definitions with physically-motivated names:

```
>>> # bad: (but I've seen much worse!)
>>> s.velocity=s.velocity+(vector(0,-9.8,0)-0.2*s.velocity)*dt

>>> # good:
>>> gamma = 0.2 # 1/s
>>> g = vector(0,-9.8,0) # meter per second^2
>>> s.acceleration = g - gamma*s.velocity # m/s^2
>>> s.velocity = s.velocity + s.acceleration*dt # m/s
```

[0] or equivalently, in radians

1 Introduction to Python

Python[1] is a general-purpose, high-level programming language, which is widely used in scientific computations when performance is not a factor. This course will be taught using the python programming language. Most problems are expected to be solved by writing python programs, and examples will be given in python. We will also use the *vpython* package[2], which provides exceptionally easy realtime three-dimensional graphics.

1.1 Running python using idle

As a calculator

Python is an interpreted programming language, which means that you can run python interactively, just like you use your calculator. One easy way to run python is to use the program `idle`, which you may open from the Applications/Programming menu. When you start `idle`, you will see a bunch of text, leading up to a prompt that looks like:

```
>>>
```

This is the python prompt. In this text, python code will be indicated with a python prompt, but you will not want to type the "`>>>`" when entering it into python. You may type any python expression at the python prompt, and when you hit "return", python will evaluate this expression an show you the result. Try typing

```
>>> 1+2
```

and see what the result is—it shouldn't be surprising. Similarly, try multiplying (`1+2*2`) and exponentiating (`1+2**3`).

Saving a program in a text file

While it is very convenient to run python interactively, it is also often helpful—for instance, when turning in homework assignments—to be able to save your

[1] For more information on python, including links to numerous tutorials, see `http://www.python.org`

[2] For more information on vpython, see `http://vpython.org`

work, edit it at your leisure, and run it again. Naturally, python supports this type of work pattern. There are numerous editors that you can use for your python programs, but one of the simplest is `idle`, which we can also use to explore python interactively. You can start a new file by selecting "New Window" in the file menu of `idle`. The file is actually created when you save it. Let's create a file named `problem-1.1.py` by creating a new window and then saving it. [3], try entering an expression such as

```
1+2*3
```

Now save the file, and in run your new program by selecting "Run Module" from the Run menu. You actually won't see anything happen. The reason is that when you run python non-interactively, python doesn't print the values of any expressions you might write. In order to convince python to print the value, you will need to use the `print` function:[4]

```
print(1+2*3)
```

When you run this program, you will need to look at the "Python Shell" window in order to see the output. You could alternatively run your python script from the bash shell (see Appendix A), but for this class you will not need to use the bash shell.

You can also group several statements in sequence, as in

```
print("Hello world!")
print(1+2*3)
print("Two to the third is")
print(2**3)
```

1.2 Imports

Python has many *modules* which add extra functions and functionality. For instance, if you try running

```
>>> sin(pi)
```

you will get an error message, indicating the the function 'sin' is undefined. The problem isn't that python is ignorant of basic trigonometry, but rather that these functions are hidden until you import them. In this class, we will almost exclusively be using functions that are exported by the `visual` module, which you can access by typing:

[3] There will always be another window, labelled something like "Python Shell", which you can use to run python code interactively.

[4] The `print` function is actually a built-in statement, but you can use it like a function for the purposes of this class. This will change in python 3, so that print *will* be an ordinary function, as it ought to be.

```
>>> from visual import *
>>> sin(pi)
1.2246467991473532e-16
```

Now we have successfully computed the $\sin(\pi)$[5]. Ordinarily, one would import from `math`, but the `visual` module re-exports numerous useful functions from other modules, and throughout this course we will simply import `visual`.

There is one more import you should make, which is

```
>>> from __future__ import division
```

This changes the divide operator so that it always behaves as ordinary division.

1.3 Comments

Any text on a line following a `#` ("pound") sign is ignored by python, and is what we call a *comment*. Comments are written to benefit any humans who might want to read a computer program.

To illustrate a comment, let's write our first 3D program:[6]

```
>>> from visual import *
>>> from __future__ import division
>>> box() # This creates a 3D cube!
```

You can now use the right mouse button to rotate this box. This is not much of a 3D program, but it's also rather compact.

In this class you will be often expected to add comments clarifying the meaning of your code—in particular, the units of your numerical constants.

1.4 Variables

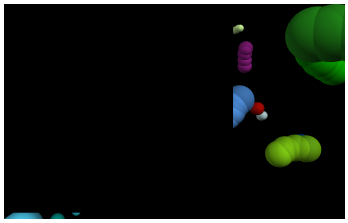
A variable in programming is very similar to the mathematical concept of a variable. It is essentially a name associated with an object or value. Variables are created using the `=` (assignment) operator, which is also used to modify the state of variables.

```
>>> x = 1
>>> y = 10
```

[5] You might be wondering how successful we were, since the exact answer for $\sin(\pi)$ is 0. We got a non-zero answer due to *roundoff error*, which is due to the limited number of *bits* the computer uses to store numbers. Although many of the practices of computational science are related to the need to mitigate the effects of roundoff errors, in this course we will gloss over these issues, which will be covered in detail in subsequent courses.

[6] You can either run this interactively, or save it in a text file. From this point on, python code in this book will always be annotated with `>>>` so that you can easily distinguish python code from its expected output—or from shell commands. Also note that in future, the `from visual import *` and `from __future__ import division` will be omitted from all program listings.

```
>>> print(x/y)
0.1
```

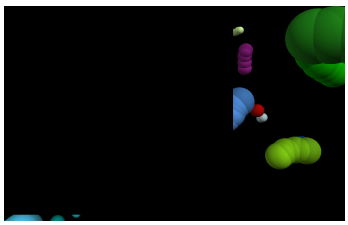


Using variables, we are able to modify objects. For instance, we can change the color of a box (try this interactively):

```
>>> cube = box()
>>> cube.color = color.green
>>> cube.color = color.blue
```

As you can see in the margin figure (possibly the most boring figure ever!), we can change the color of a cube. The '.' operator allows us to modify properties of objects. VPython defines a number of properties, such as `color`, `pos` (position). Play a bit with the following examples.

```
>>> floor = box()
>>> floor.color = color.green
>>> floor.size = (2,0.2,2)
>>> ball = sphere()
>>> ball.color=color.red
>>> ball.radius=0.55
>>> ball.radius = 0.6
```



You can also set properties when you create an object.

```
>>> disk = cylinder(color=color.yellow, radius=sqrt(2)/2, length=0.1)
```

1.5 Assignment semantics

In python, the = operator behaves in one of two ways. If the left hand side is a simple variable, as in

```
>>> ball = sphere()
```

the left hand side (`ball` in this case) is set to be a name for the object on the right hand side (which is in this case an anonymous sphere). In contrast, if the left hand side is a sub-object, then the object referenced on the left hand side is actually modified. So

```
>>> ball.color = color.blue
```

doesn't just create a new name for the object `color.blue`, but actually modifies the existing object which is called `ball`.

It may sound pretty confusing, but it's actually quite simple, as hopefully the following examples will illustrate.

Problem 1.1 Standard assignment semantics Try to predict the result of the following program, and then execute it interactively.

```
>>> a = 10
>>> b = a
>>> b = 0
>>> print(a)
```

What is the final value of `a`? Why?

Problem 1.2 Assignment to objects Try to predict the result of the following program, and then execute it interactively. What is the final value of `a.taste`? Why, and how does this differ from the result of Problem 1.1?

```
>>> a = sphere()
>>> a.taste = "sour"
>>> b = a
>>> b.taste = "sweet"
>>> print(a.taste)
```

Assignment to and from vpython objects

Alas, the above description of python's assignment semantics does not fully apply to *visual* python, which is an entirely different beast. Try to guess the value of `b` in the two programs below:

```
>>> a = sphere()                  >>> a = sphere()
>>> a.pos = vector(0,0,0)         >>> a.foo = vector(0,0,0)
>>> b = a.pos                     >>> b = a.foo
>>> a.pos = vector(1,1,1)         >>> a.foo = vector(1,1,1)
>>> b                             >>> b
```

The trouble here is that the data member `a.pos` is a native vpython data member, and is treated specially—and in an unhelpful manner at that! To avoid this confusion, *never place a vpython native field alone on the right hand side of an assignment.* One option for avoiding this is to multiply vectors by one. Another option is to wrap them in a constructor function, as in

```
>>> b = vector(a.pos)
```

1.6 Functions

In this course, we will rarely define functions, since we'll be writing quite simple programs. However, functions are at the core of all "real" programs. A function is defined as

```
>>> def add(x,y):
...   return x + y
```

Thes defines a function called `add`, which accepts two arguments, which we call `x` and `y`. The value of this function is just the sum of its arguments. The `return` statement both exits the function, and determines its value. You can call this function with code such as

```
>>> add(1,2)
```

Function definitions demonstrate an interesting feature of python, which is its white space determined block structure. A *block* begins with a line ending with a colon, and is indented by some amount. The block continues as long as the indentation remains the same. In this manner, you can define functions that extend over multiple lines. The following (foolish) example demonstrates a multi-line function definition.

```
>>> def add(x,y):
...   a = x
...   b = y
...   return a + b
```

Note on lexical scoping

The term *lexical scoping* refers to which parts of the code can "see" the names of variables. By default, variables in python are *locally scoped*, meaning that they are visible only in the function in which they are defined. This means that each function can be read and understood independently, since even if the names `x`, `y`, `a` and `b` are used elsewhere in the code, the above function doesn't refer to the *same* `x`, `y`, `a` and `b`.

1.7 Conditionals

Functions would be pretty boring if they always did the same thing. The `if` statement allows you to write code that does something different based on some condition.

```
>>> a = 1
>>> if a > 0:
...   print("It's greater than zero!")
```

Note that the `if` statement, like function definitions, uses a *block* determined by indentation.

Of course, at times you want to do something in either case, in which case you want the `else` option. You can try the following, which writes a program

```
>>> def announce(a):
...   if a > 0:
...     print("It's positive!")
...   else:
...     print("It's not positive!")
```

Finally, you can nest if statements, as in:

```
>>> def announce(a):
...     if a > 0:
...         print("It's positive!")
...     else:
...         if a == 0:
...             print("It's exactly zero!")
...         else:
...             print("It's negative!")
```

math	python
$=$	`==`
$>$	`>`
$<$	`<`
$\geq$	`>=`
$\leq$	`<=`
and	`and`
or	`or`
not	`not`

Boolean operators

Boolean expressions

In order to effectively use the `if` statement, we need to be able to express values that are `true` or `false`. These are called *booleans*. Although you can explicitly specify boolean values by typing `true` or `false` into your program, most often you will use the comparison operators you're familiar with from math, which are shown in the margin.

Problem 1.3 minimum(a,b) Write a function that returns the minimum of its two arguments.

Problem 1.4 Is a divisible by b? Write a function that determines if one of its arguments is evenly divisible by the other. You may want to use the standard library function `round` which rounds a number to the nearest integer.

1.8 Looping

In programming, *loops* are constructs that allow us to do something repetitively. Loops are at the core of most programs simply because repetitive work is what most of us don't like to do by hand, but computers are very good at (because they don't get bored). There are several ways to create a loop in python, but in this course, we will use only one of them, the *while* loop.

```
>>> i = 0
>>> while i < 10:
...   print(i)
...   i = i + 1
```

This example is a pretty standard sort of loop. We first create a variable initialized to 0. Then we begin the `while` loop. A `while` loop has a boolean expression, and it keeps executing while that expression is true. Next comes a block of code, which ought eventually to make the expression true, otherwise you've created an infinite loop!

The last line in this example renames the `i` variable to the value `i+1`. This is an extremely common idiom, in which we update a value during a loop. In

fact, it's so common that python (and many other languages) has a special operator for this.

Problem 1.5 Identifying primes Write a function that given an integer `n` determines if it is prime. There are a number of algorithms you can use, some more efficient than others, but for now can simply check using your function from Problem 1.4 whether `n` is divisible by any integers smaller than itself but greater than 1.

Problem 1.6 = versus == What is the difference between the following two statements?

```
>>> i = i + 1
```

```
>>> i == i + 1
```

Problem 1.7 Factorial Implement a factorial function, which computes $n!$, which is the product of all integers from 1 to n. The factorial of 0 is 1.

1.9 Recursion

Finally, we come to recursion, which is when a function is defined in terms of itself. We won't often be using recursion in this class, but it's useful, so I'll mention it here very briefly. As an example, I will introduce a solution to Problem 1.4 that does *not* use either the round function or division. It's also extremely slow...

```
>>> def isdivisible(a,b):
...   if a == b:
...     return True
...   if a > b:
...     return isdivisible(a-b,b)
...   return False
```

Problem 1.8 Factorial II Implement your factorial function using the recursion relation that if $n > 0$ then $n! = n(n-1)!$ and $0! = 1$.

2 Position—the simplest vector

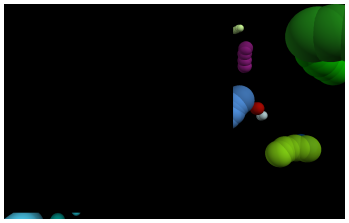
The meaning in Physics of *position* (closely related to *displacement*) is almost identical to the every-day usage. The primary distinction is that by position, we mean only the location of the center of a point, not the orientation of a body. Position is described by a *vector*, which must be specified relative to some coordinate system. A *displacement* is the difference between two positions, which is also a vector. Both position and displacement have units of distance, which is *meter* in the SI system of units.

2.1 What is a vector?

A *vector* is a quantity that is completely specified by a direction and magnitude. Vectors lie at the core of a physicist's description of the world around us. In this text, vectors are typeset in bold face, as in $\mathbf{r}$ or $\mathbf{v}$. When you write vectors by hand, you will write them as $\vec{r}$ or $\vec{v}$. The *position* is commonly written using the symbol $\mathbf{r}$.

We commonly define vectors relative to some coordinate axes, which we will call $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$. The common notation uses the "hat" symbol "^" to represent a *unit vector*. A unit vector is a vector with magnitude of one. Together, $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ form a *basis set*,[1] meaning that we can describe any vector as a sum of these three. Thus, we will often write the position vector as

$$\mathbf{r} = r_x\hat{\mathbf{x}} + r_y\hat{\mathbf{y}} + r_z\hat{\mathbf{z}} \tag{2.1}$$



where r_x, r_y and r_z are the *components* of the vector $\mathbf{r}$. You can think of them as the coordinates on a three-dimensional grid. Unit vectors are inherently unitless, so the units of the components r_x, r_y and r_z of a position vector must be the same as the units of the position vector itself, which is meters—in SI units.

A *scalar* is a quantity that has a magnitude but no direction. Moreover, a scalar is defined to be a quantity whose value does not change when the coordinate system is rotated. We typeset scalar quantities in italics, as in r or t. Common examples of scalars include quantities like *distance*, *radius*, *time* or *energy*.

[1] In particular, they form an *orthonormal basis set*—the best kind—but we won't go into any more detail on that just yet.

2.2 Vector operations

Vectors can only interact with scalars in a limited number of ways. Knowing this allows us to easily catch certain sorts of errors.

The *magnitude* of a vector is a scalar. We can compute the magnitude of a vector using the Pythagorean Theorem:

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2} \tag{2.2}$$

Two vectors that have the same units may be *added* or *subtracted*. Addition and subtraction affect both the

$$\mathbf{a} = \mathbf{b} + \mathbf{c} \tag{2.3}$$

$$a_x\hat{\mathbf{x}} + a_y\hat{\mathbf{y}} + a_z\hat{\mathbf{z}} = b_x\hat{\mathbf{x}} + b_y\hat{\mathbf{y}} + b_z\hat{\mathbf{z}} + c_x\hat{\mathbf{x}} + c_y\hat{\mathbf{y}} + c_z\hat{\mathbf{z}} \tag{2.4}$$

$$= (b_x + c_x)\hat{\mathbf{x}} + (b_y + c_y)\hat{\mathbf{y}} + (b_z + c_z)\hat{\mathbf{z}} \tag{2.5}$$

$$a_x = b_x + c_x \tag{2.6}$$

A vector may be multiplied by a scalar—this is referred to as *scalar multiplication*. In this case, the direction is unchanged, while its magnitude (and possibly units) are changed.

$$\mathbf{F} = m\mathbf{a} \tag{2.7}$$

In addition, there are two ways that pairs of vectors may be multiplied. The *dot product* is a multiplication of two vectors that results in a scalar. The dot product of any vector with itself is the square of its magnitude.

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z \tag{2.8}$$

Two vectors are said to be *orthogonal* if their dot product is zero.

Finally, the cross product takes two vectors and produces a third vector.[2]

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - b_z a_y)\hat{\mathbf{x}} + (a_z b_x - a_x b_z)\hat{\mathbf{y}} + (a_x b_y - a_y b_x)\hat{\mathbf{z}} \tag{2.9}$$

The cross product of two parallel vectors is zero, and the result of a cross product of two vectors is orthogonal to either of those vectors.

$$\mathbf{a} \times \mathbf{a} = 0 \tag{2.10}$$

$$\mathbf{a} \cdot (\mathbf{a} \times \mathbf{b}) = 0 \tag{2.11}$$

$$\mathbf{b} \cdot (\mathbf{a} \times \mathbf{b}) = 0 \tag{2.12}$$

$$\tag{2.13}$$

In addition, the cross product is odd when commutated, meaning that

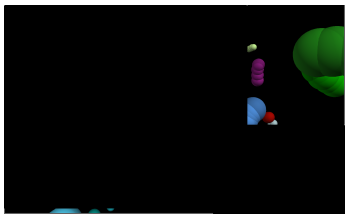
$$\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a} \tag{2.14}$$

[2]Technically, the result of a cross product is a *pseudovector*, because it changes direction when the coordinate system is inverted.

2.3 Putting vectors into python

In visual python, you can create a vector from its components as follows:

```
>>> r = vector(1,0,0) # This is a vector in the x direction
>>> arrow(axis=r, color=color.blue)
```



Here we visualize the vector by creating an arrow. Of course, we can also specify the vector directly in the arrow function:

```
>>> arrow(axis=vector(1,0,0), color=color.blue)
```

We can compute the magnitude of a vector using the `abs`[3] function, which is also used to compute the absolute value of a scalar.

```
>>> abs(vector(1,1,1))
```

We can perform scalar multiplies using the `*` operator as we would for ordinary multiplication.

```
>>> r = vector(1,0,0) # in meters
>>> s = -3.0
>>> arrow(axis=r, color=color.blue)
>>> arrow(axis=s*r, color=color.red)
```

Here you can see an annoyance of the `arrow` object in visual python: when we scale its length, its diameter scales as well. We can rectify this by specifying the diameter manually:

```
>>> r = vector(1,0,0) # in meters
>>> s = -3.0
>>> arrow(axis=r, color=color.blue, shaftwidth=0.1)
>>> arrow(axis=s*r, color=color.red, shaftwidth=0.1)
```

The `axis` of an `arrow` defines the position of its head relative to that of its tail. The position of its tail is defined by its `pos`, so we can shift our arrows by adjusting their `pos`

```
>>> v1 = vector(1,0,0) # in meters
>>> v2 = -2*v1 # meters
>>> a1=arrow(axis=v1, color=color.blue, shaftwidth=0.1)
>>> a2=arrow(axis=v2, color=color.red, shaftwidth=0.1)
>>> a1.pos = vector(-1.5,1,0) # meters
```

[3] You can also use the `mag` function to compute the magnitude of a function, but this function will only work on vectors, so by remembering `abs` you remember two birds with one stone...

Problem 2.1 Vector arithmetic Work out by hand what the output of the following python programs should be, and draw a picture of what output you expect. After showing the instructor your predictions, try running the programs, and see how your predictions compared.

Addition of two vectors

```
>>> a = vector(1,2,0)
>>> b = vector(2,0.5,0)
>>> arrow(axis=a, color=color.blue)
>>> arrow(axis=b, color=color.red)
>>> arrow(axis=a+b)
```

Subtraction of two vectors

```
>>> a = vector(1,2,0)
>>> b = vector(2,2,0)
>>> arrow(axis=a, color=color.blue)
>>> arrow(axis=b, color=color.red)
>>> arrow(axis=a-b)
```

Multiplication of two vectors

```
>>> a = vector(1,2,0)
>>> b = vector(2,2,0)
>>> arrow(axis=a, color=color.blue)
>>> arrow(axis=b, color=color.red)
>>> arrow(axis=a*b)
```

Cross product of two vectors

```
>>> a = vector(1,2,0)
>>> b = vector(2,2,0)
>>> arrow(axis=a, color=color.blue)
>>> arrow(axis=b, color=color.red)
>>> arrow(axis=cross(a,b))
>>> arrow(axis=cross(b,a), color=color.yellow)
```

Dot product of two vectors

```
>>> a = vector(1,2,0)
>>> b = vector(2,2,0)
>>> arrow(axis=a, color=color.blue)
>>> arrow(axis=b, color=color.red)
>>> print(dot(a,b))
>>> arrow(axis=dot(a,b))
```

Problem 2.2 Making a box out of cylinders The cylinder object in VPython is created using code such as

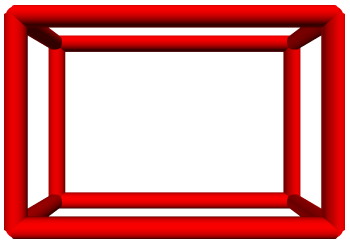
```
>>> cylinder(pos=vector(10, 0, 0), axis=vector(10,0,0))
```

Cylinders are constructed using two vectors, the `pos` and the `axis`. The `pos` describes the location of one end of the cylinder, while the `axis` is the displacement vector from the first end of the cylinder to the other end.

Make a box with corners at $\pm x_{max}\hat{\mathbf{x}} \pm y_{max}\hat{\mathbf{y}} \pm z_{max}\hat{\mathbf{z}}$, where the constants x_{max}, y_{max} and z_{max} are defined by

```
>>> xmax=15
>>> ymax=10
>>> zmax=5
```

As a challenge, you can use `for` or `while` loops to reduce the number of times `cylinder` occurs in your code to a minimum.



Problem 2.2 Making a box out of cylinders

2.4 Moving things around

Just placing balls on the screen isn't very exciting. Let's start moving things around. To start with, let's make a ball move on one direction with constant velocity. In this class, you must *always* add a comment defining the units of any constants you define. Units provide an important check on the correctness of physical equations, be they code or formulas (this is also known as "dimensional analysis").

```
>>> s=sphere()
>>> t = 0 # seconds
>>> dt = 0.01 # seconds
>>> v = 10 # meters/second
>>> while t < 1: # second
...   s.pos = vector(v*t,0,0) # meters
...   t = t+dt
```

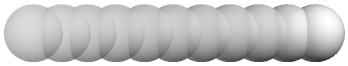
The first thing you will notice with this code is that it finishes much too rapidly to see what has happened. To slow it down, we add a call to the `rate` function, which ensures that our loop happens no more than $1/\Delta t$ times per second.

```
>>> while t < 1: # second
...   s.pos = vector(v*t,0,0) # meters
...   t = t+dt
...   rate(1/dt)
```

Even with the `rate` function, the motion is pretty confusing to follow. The reason is that visual python automatically zooms the camera so as to keep all the objects in view—while it remains pointed at the origin. This is often a helpful feature, but doesn't really help when there is just one object to be seen.

There are two solutions to this dilemma. One is to tell vpython how big you want the field of view to be, and to instruct it not to change this field. We can do this with:

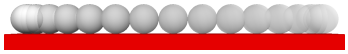
```
>>> scene.range = 10 # meters
>>> scene.autoscale = False
>>> s=sphere()
>>> t = 0 # seconds
>>> dt = 0.01 # seconds
>>> v = 10 # meters/second
>>> while t < 1: # second
...   s.pos = vector(v*t,0,0) # meters
...   t = t+dt
...   rate(1/dt)
```



Another option, which is often nicer—particularly when your objects unexpectedly depart from their intended trajectory—is to add other objects to provide a stationary frame of reference. One such option would be a large slab:

```
>>> ground=box(color=color.red)
>>> ground.size = (22,1,2) # meters
>>> ground.pos = vector(0,-1.5,0) # meters
>>> s=sphere()
>>> t = 0 # seconds
>>> dt = 0.01 # seconds
>>> v = 10 # meters/second
>>> while t < 1: # second
...   s.pos = vector(v*t,0,0) # meters
...   t = t+dt
...   rate(1/dt)
```

Problem 2.3 Sinusoidal motion Write a program to make your ball move with sinusoidal motion in one dimension. As you will learn later, this is the motion of a simple harmonic oscillator (fancy Physics talk for a spring).



2.5 Circular motion

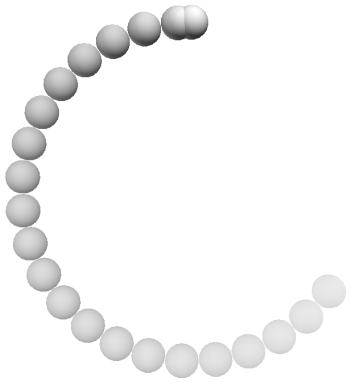
Now let's consider uniform circular motion. Circular motion is most easily expressed using polar coordinates. The azimuthal angle θ varies linearly with time, such that

$$\theta(t) = \omega t \tag{2.15}$$

where ω is the *angular speed*, which has units of *radians per second*. The radius r remains constant. Expressing the displacement in cartesian coordinates, we obtain:

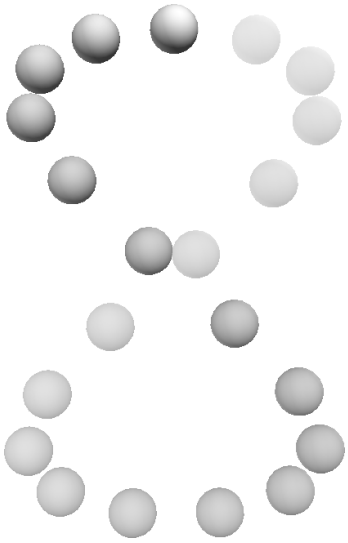
$$\mathbf{r}(t) = R\sin(\omega t)\hat{\mathbf{x}} + R\cos(\omega t)\hat{\mathbf{y}} \tag{2.16}$$

```
>>> s=sphere()
>>>
>>> t = 0 # seconds
>>> dt = 0.001 # seconds
>>> R = 10 # meters
>>> omega = 2 # radians/second
>>> while t < 2*pi/omega:
...   s.pos = vector(R*sin(omega*t),R*cos(omega*t),0)
...   t = t+dt
...   rate(1/dt)
```



Problem 2.4 Figure eight Modify the circular motion from the previous section to cause the ball to move in a figure eight pattern, as depicted in the figure.

Figure 2.1: Problem 2.4 Figure eight



3 Velocity—making things move

The *velocity* is the *derivative* of position with respect to time.

$$\mathbf{v}(t) \equiv \frac{d\mathbf{r}(t)}{dt} \tag{3.1}$$

At this point, it is worth noting that it is customary to omit the parentheses reminding us that position and velocity are functions of time, so that Equation 3.1 would commonly be written as

$$\mathbf{v} = \frac{d\mathbf{r}}{dt} \tag{3.2}$$

This is a mathematical definition of velocity, but what does velocity *mean*? To answer that question, let us explore the question of what a *derivative means*.

3.1 What is a derivative?

Mathematically, a derivative is defined as

$$\frac{dy}{dx} \equiv \lim_{\Delta x \to 0} \frac{y(x + \Delta x) - y(x)}{\Delta x}, \tag{3.3}$$

and if this limit is well-defined, the function $y(x)$ is called *differentiable*.[1] Note that in this equation, we use the greek letter Δ to mean "*a change in*". This is a common and customary usage. The numerator is also a change: it is the change in y when x changes by an amount Δx. So we can crudely write that

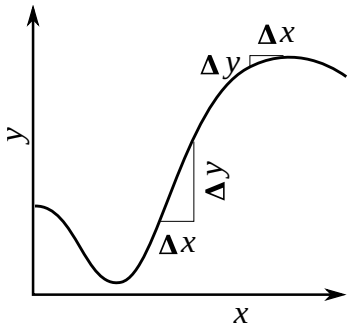
$$\frac{dy}{dx} = \lim_{\Delta \to 0} \frac{\Delta y}{\Delta x} \tag{3.5}$$

In other words, the derivative $\frac{dy}{dx}$ is a *a little change in y* divided by a *a little change in x*. One helpful way of thinking of this is as the slope of the plot

[1] In fact, the Equation 3.3 is not always the most useful definition for a derivative. One common definition is the *centered* derivative:

$$\frac{dy}{dx} \equiv \lim_{\Delta x \to 0} \frac{y(x + \Delta x/2) - y(x - \Delta x/2)}{\Delta x} \tag{3.4}$$

In the limit as $\Delta x \to 0$, it is evident that these two definitions are equivalent for any smooth function. The advantage of Equation 3.4 accrues when we choose to approximate this derivative using a small but nonzero value for Δx.



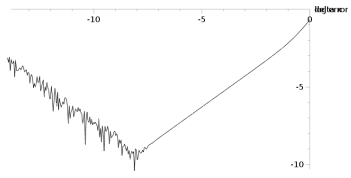
of y versus x, since $\frac{\Delta y}{\Delta x}$ is just that slope. However, when y is a vector (as is the case for the velocity), the interpretation of its derivative as a simple slop is complicated.

Thus the velocity $d\mathbf{r}/dt$ is *a little change in position divided by the corresponding little change in time*. The velocity is the rate of change of position, or how fast the position is changing with time, and in what direction. Velocity is a vector with units of meter/second. The magnitude of the velocity is called the *speed*. Thus *speed* is a scalar, while *velocity* is a vector.

3.2 The finite difference method

The *finite difference method* is an approach for numerically evaluating a derivative, and corresponds to simpley evaluating Equation 3.3 for some finite value of Δx. This is the standard method for computing a numerical derivative, although it is relatively uncommonly used, simply because it is usually easy to work out derivatives analytically. It does, however, often find use as a means of checking the correctness of an analytically-computed derivative. In the following example, we plot the error introduced when performing a finite-difference derivative of $\sin(x)$, using our knowledge that the derivative of this function is actually $\cos(x)$

```
>>> from visual.graph import * # 2D graphing
>>> gdisplay(xtitle='delta x', ytitle='log error',
...          foreground=color.black, background=color.white)
>>> error = gcurve()
>>>
>>> # The following is a finite-difference derivative of sin(x)
>>> def dsin(x,dx):
...   return (sin(x+dx) - sin(x))/dx
>>>
>>> x = 0.1 # radians
>>> dx = 1 # radians
>>> while dx > 1e-14: # radians
...   error.plot(pos=(log10(dx),log10(abs(dsin(x,dx)-cos(x)))))
...   dx = dx/1.1
```



As you can see in the generated plot, the error decreases as we decrease the size of Δx—but only up to a point. It makes sense that the error should decrease as we decrease Δx, since the derivative is defined to be the limit as Δx approaches zero. However, when Δx becomes *too* small (around 10^{-7} in this case), *roundoff* causes our error to begin increasing.

Problem 3.1 The centered finite difference method Repeat the exercise in Section 3.2 using the central difference approach described in Equation 3.4. How does the this method compare with the off-center derivative? Which approach is more accurate? How does the error depend on the size of Δx?

Hint: You may find it helpful to compare the accuracty of each approch using two values of Δx such as 0.1 and 0.01.

3.3 Integration—inverting the derivative

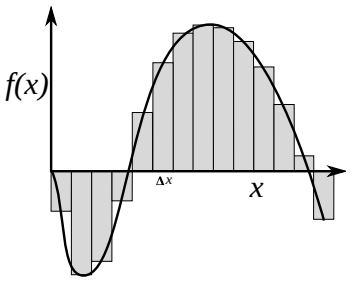
Suppose we know the velocity of an object, how would we determine the position? Common sense tells us that we must know more in order to answer that question: we need to know the objects initial position at some time.

Suppose you know the derivative of a function, how would you find the function itself? This is particularly relevant in this chapter, as we want to be able to determine the position of an object based on a knowledge of its velocity. The general answer is that we must perform an *integral* in order to compute a function from its derivative.

A definite integral—which is the only sort that we can perform numerically—may be defined as:

$$\int_a^b f(x)\ dx \equiv \lim_{\Delta x \to 0} \sum_{x=a}^{b} f(x)\Delta x \tag{3.6}$$

where it is understood that the summation over x values on the right hand side is performed over values of x that differ by Δx. The integral can also be seen as giving the *area* under the curve plotting $f(x)$, as illustrated to the right, provided it is understood that when the function is negative, it contributes negative area.



Quadrature is the process of numerically solving a definite integral.

The definition of the integral also clarifies the notation we use for integrals. The "$\int$" symbol is a stretchesd "S" standing for "summation". The dx is the limit of a very small Δx, just as we saw in the definition of the derivative. As the derivative describes a change—in other words, a difference—the integral describes a summation of little changes.

The definition of an integral in Equation 3.6 immediately suggests an algorithm for numerically computing an integral. We simply need sum over many points separated by Δx. And indeed, this is at the heart of any integration scheme. The details of quadrature methods such as the midpoint method, the trapezoidal method or Simpson's rule all come down to approaches for dealing with the end points, an issue that is irrelevant in the limit as $\Delta x \to 0$.

To get back to the problem of solving for position when we only know the velocity, the *fundamental theorem of calculus* states that

$$\int_a^b \frac{dy}{dx}\ dx = y(b) - y(a) \tag{3.7}$$

This theorem states what we pointed out at the beginning of this section, which is that if we know the time derivative of the position, we can integrate that value in order to find the position at any time, provided we know the initial position.

3.4 Euler's method

Euler's method is a time-honored approach for integrating, which is useful when we want to compute many integrals of the same function, each with different bounds.[2] Euler's method is an approach to solve the equation

$$f(t) = f(0) + \int_0^t \frac{df(\tau)}{d\tau} d\tau \tag{3.8}$$

for a sequence of values for t, assuming that we can compute $\frac{df}{dt}$ and already know $f(0)$. Here I've chosen to introduce a dummy variable τ for the integration, in order to distinguish between the time t at which we're computing the function $f(t)$ and those intermediate times that will arrise in the summation that describes the integral.

The simples way to arrive at Euler's method is to consider the definition of a derivative:

$$\frac{df}{dt} \equiv \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}, \tag{3.9}$$

If we pick a small value for Δt instead of taking the limit, then we can solve this equation for $f(t + \Delta t)$:

$$f(t + \Delta t) \approx f(t) + \frac{df}{dt} \Delta t \tag{3.10}$$

With this relationship, if we know $f(0)$ we can compute $f(\Delta t)$ and from that $f(\Delta t + \Delta t)$ and so on. One question that remains is at what time the derivative $\frac{df}{dt}$ should be evaluated. Euler's method uses the pragmatic choice of evaluating the derivative at the earlier time t:

$$f(t + \Delta t) \approx f(t) + \left.\frac{df}{dt}\right|_t \Delta t \tag{3.11}$$

Other options would have been to evaluate the derivative at the later time $t + \Delta t$, or at the central time $t + \Delta t/2$.

How does Euler's method relate to integration, as defined in Equation 3.6? The key is to recognize that as we repeatedly apply Equation 3.11, we are repeatedly summing $\frac{df}{dt} \Delta t$—which is precisely the definition of an integral, in the limit that Δt is small. So although we motivated Euler's method by considering the definition of a derivative, we ended up with an evaluation of an integral. This is due to the *fundamental theorem of calculus*.

3.5 Constant velocity

To illustrate the use of Euler's method, we will first consider the case of a ball moving with *constant velocity*. We will begin by creating a box to be the "ground" in which ball moves.

[2] Euler's method also allows the integration of systems of partial differential equations, but we will leave that application for Chapter 4.

```
>>> ground=box(color=color.red)
>>> ground.size = (22,1,2) # meters
>>> ground.pos = vector(0,-1.5,0) # meters
```

Then we create a sphere with an initial position and velocity. Note that although velocity is not directly supported by `sphere` objects, python allows us to assocate any data we wish with any object. We'll start the ball out on the left side, and make its velocity be to the right with a speed of 10 meters per second.

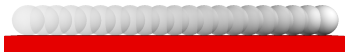
```
>>> s=sphere()
>>> s.pos = vector(-10,0,0) # meters
>>> s.velocity = vector(10,0,0) # meters per second
```

Next define an initial time, and a Δt. It doesn't greatly matter what value we pick for Δt, as long as it is reasonably small. In this case, "small" can be defined by considering how far the ball will move in one step, which is $v\Delta t$. Since the scene (defined by the ground) is 22 m long, we want to ensure that $v\Delta t \ll 22$ m, or $\Delta t \ll 0.5$ s.

```
>>> t = 0 # seconds
>>> dt = 0.01 # seconds
```

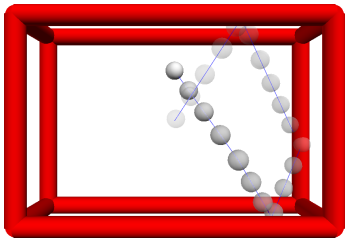
Finally we reach the actual Euler's method integration. We need to loop over t, incrementing by Δt. We can achieve this with a `while` loop, very similar to the loops we used in the last chapter.

```
>>> while t < 2:
...     s.pos = s.pos + s.velocity*dt
...     t = t+dt
...     rate(1/dt)
```



As you can see, Euler's method is actually very simple to implement.

Problem 3.2 Ball in a box I Write code to modify the ball's motion to make it bounce around inside a box. This will involve inserting into the while loop checks to see if the $\hat{\mathbf{x}}$ position of the ball is outside the bounds and whether its velocity indicates that it is leaving the box. If this happens, you will want to reverse the sign of the $\hat{\mathbf{x}}$ component of the velocity. You will probably want to do similarly for the $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ components. Note that you can access the $\hat{\mathbf{x}}$ component of the vector `s.velocity` as `s.velocity.x`, and similarly for the $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ components.



Problem 3.2 Ball in a box I

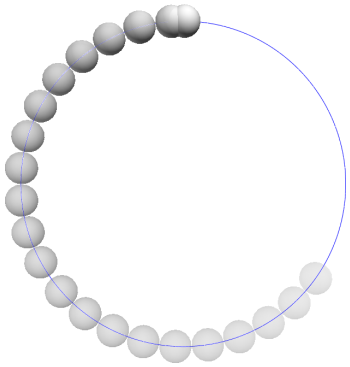
It would be best to use your solution to Problem 2.2, so you can see that the ball says within the bounts set. If you didn't solve that problem, you may use the solution below to draw a box around your bounds.

```
>>> xmax=15
>>> ymax=10
```

```
>>> zmax=5
>>> for x in [-xmax,xmax]:
...   for y in [-ymax,ymax]:
...     cylinder(pos=vector(x, y, -zmax),
...              axis=vector(0,0,2*zmax),
...              color=color.red)
>>> for x in [-xmax,xmax]:
...   for z in [-zmax,zmax]:
...     cylinder(pos=vector(x, -ymax, z),
...              axis=vector(0,2*ymax,0),
...              color=color.red)
>>> for y in [-ymax,ymax]:
...   for z in [-zmax,zmax]:
...     cylinder(pos=vector(-xmax, y, z),
...              axis=vector(2*xmax,0,0),
...              color=color.red)
```

Problem 3.3 Circular motion revisited In Section 2.5, we made a ball move in a circular by setting its position according to Equation 2.16. Create this same motion by using Euler's method to integrate the velocity, which you can determine by taking the derivative of $\mathbf{r}(t)$. Note: to take a derivative of a vector-valued function such as $\mathbf{r}(t)$, you can simply take a derivative of each of the components of $\mathbf{r}$ individually.

Figure 3.1: Problem 3.3 Circular motion revisited



Problem 3.4 Figure eight revisited Work out a velocity $\mathbf{v}(t)$ which will lead to motion in a figure-eight pattern (as illustrated in Problem 2.4), and program this motion using Euler's method.

Problem 3.5 Guided missiles Consider four missiles initially located at the four corners of a square with a side of 100m. Model their behavior if each

missle moves with a speed of 5m/s in the direction pointing directly at the missile counter-clockwise from itself.

Problem 3.5 Guided missiles

4 Acceleration and forces—kinematics and dynamics

4.1 A second derivative

The *acceleration* is defined as the rate of change of velocity

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} \tag{4.1}$$

From this definition, you can see that the SI units for acceleration are $\mathrm{m/s^2}$. As we shall see (in Section 4.3), the acceleration is directly related to the forces acting on an object, and as such plays a central role in Newtonian mechanics.

You can *feel* acceleration acting upon you as if it were a change in the gravitational force. When you go into free fall on a roller-coaster, you feel weightless, as if there were no gravitational force. Similarly, when a car turns at high speed—which involves acceleration, since the velocity is changing—you feel a force to the side. The term *G-force* is a measure of acceleration, measured relative to the acceleration that gravity induces on objects in free fall. An object in free fall near the Earth's surface accelerates downward at a rate of $g = 9.8\ \mathrm{m/s^2}$.

From the definition of velocity, we can see that the acceleration is the second derivative of position:

$$\mathbf{a} = \frac{d^2\mathbf{r}}{dt^2} \tag{4.2}$$

How do we solve for the position $\mathbf{r}$, knowing the acceleration? Since a second derivative is a derivative of a derivative, one obvious idea would be to take an integral of an integral. This will be our first approach, which is still called Euler's method.

4.2 Euler's method revisited

In Section 3.4, we learned to use Euler's method to integrate the velocity $\mathbf{v}$ in order to find the position $\mathbf{r}$. Evidently, we can use the same approach to find the velocity from the acceleration $\mathbf{a}$, and then proceed as before to compute the position from the velocity. The key is that using Euler's method, these

two integrations can be performed simultaneously, with a minimum of extra book-keeping.

Here we are solving two coupled differential equations

$$\frac{d\mathbf{r}(t)}{dt} = \mathbf{v}(t) \tag{4.3}$$

$$\frac{d\mathbf{v}(t)}{dt} = \mathbf{a}(t) \tag{4.4}$$

Applying Equation 3.11 to each of these gives us two coupled equations

$$\mathbf{v}(t + \Delta t) \approx \mathbf{v}(t) + \mathbf{a}(t)\Delta t \tag{4.5}$$

$$\mathbf{r}(t + \Delta t) \approx \mathbf{r}(t) + \mathbf{v}(t)\Delta t \tag{4.6}$$

which we can implement using much the same code as we used in the last chapter.

Problem 4.1 Baseball I To make things interesting, we will implement Euler's method under constant acceleration using baseball pitching as an example. I'll set up the parameters in the code below (dimensions of the field, height of pitcher, etc), and you write the code to propagate the ball. Note: if you include a `trail.append(ball.pos)` in your `while` loop, the ball will leave a trail behind, so you can see its trajectory after it has passed.

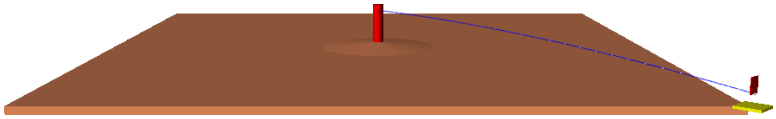
```
>>> dirt = (1,0.6,0.4) # (red,green,blue)
>>>
>>> ground=box(color=dirt)
>>> ground.axis = vector(1,0,1) # diamond orientation
>>> ground.size = (0.305*120/sqrt(2),0.305*1,0.305*120/sqrt(2)) # meters

>>> ground.pos = 0.305*vector(0,-0.5,0) # meters
>>>
>>> mound=cone(color=dirt)
>>> mound.axis= 0.305*vector(0,10/12,0) # meters
>>> mound.radius = 0.305*9 # meters
>>>
>>> pitcher=cylinder(color=color.red)
>>> pitcher.pos = 0.305*vector(0,10/12,0)
>>> pitcher.axis = 0.305*vector(0,6,0)
>>> pitcher.radius = 0.305*0.8
>>>
>>> # use a large home plate for visibility
>>> plate=box(color=color.yellow)
>>> plate.size = (1,0.1,1) # meters
>>> plate.pos = 0.305*vector(60.5,0,0) # meters
>>>
>>> strikezone = box(color=color.red)
```

```
>>> strikezone.size = (0.305/12,0.305*2,0.305*2) # meters
>>> strikezone.pos = 0.305*vector(60.5,2.5,0) # meters
>>>
>>> ball=sphere()
>>> # The pitcher's mound is 60'6" from home plate, and 10" high.

>>> # We assume the pitcher pitches from 5' above the mound
>>> ball.pos = 0.305*vector(0, 5+10/12, 0)# meters
>>> # 80 mph is the speed of an average MLB pitch
>>> ball.velocity = 0.447*vector(80,0,0) # meters per second
>>> ball.radius = 0.305*1.45/12 # meters
>>>
>>> # The ball moves pretty fast, so let's leave a trail, so
>>> # we can see if we hit the strike zone:
>>> trail = curve(color=color.blue, pos=ball.pos)
```

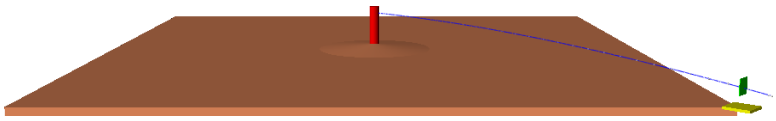
Figure 4.1: Problem 4.1 Baseball I



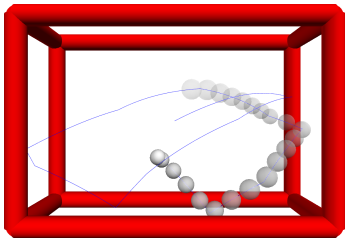
At what angle does the pitcher need to throw the ball in order to make it pass through the strike zone?

Problem 4.2 Umpire Modify your baseball program above to check whether the pitch was a strike or a ball based on whether it passes through the strike zone. In the case of a strike, turn the `strikezone` green.

Figure 4.2: Problem 4.2 Umpire



Problem 4.3 Ball in a box II Repeat Problem 3.2, but with gravity included. Try different starting velocities and starting positions, and look for patterns that emerge. What do you need to do to keep the ball from ever hitting the top of the box?



Problem 4.3 Ball in a box II

4.3 Newton's first and second laws

Newton's first law states that an object in motion tends to remain in motion, and an object at rest remains at rest, in the absence of any force upon that object. This law is clearly true only in certain reference frames, called *inertial* reference frames. In an accelerating or rotating reference frame, even objects that have no force exerted on them will appear to be changing velocity.

Newton's second law states that the acceleration of an object in an inertial reference frame is equal to the total force on that object divided by its mass. Mathematically, we write this as

$$\mathbf{F} = m\mathbf{a} \tag{4.7}$$

Note that the second law—in one sense—encompasses the first law, since when $\mathbf{F} = 0$, it means that $\frac{d\mathbf{v}}{dt} = 0$, which means the velocity is constant.

Note that Newton's second law is only approximate, as it neglects effects of relativity, which become significant when speeds approach the speed of light, 299,792,458 m/s. Fortunately, this is a rather high speed, and there are many problems that we can deal with quite effectively using Newtonian mechanics.

Problem 4.4 Circular motion Consider the problem of circular motion you treated in Problem 3.3, in which you computed the velocity corresponding to the motion described in Equation 2.16. You now know $\mathbf{r}(t)$ and $\mathbf{v}(t)$.

a. Take a derivative to find the acceleration $\mathbf{a}(t)$.

b. How does the magnitude of the acceleration relate to the magnitude of the speed? Give an explicit mathematical formula for the magnitude of the acceleration as a function of the radius and speed. This acceleration is called the *centripetal acceleration*.

c. What is the force needed—specify both magnitude and direction—in order to maintain circular motion? This force is called the *centripetal force*.

d. What is the angle between the force and the velocity? You can compute this using the relation $\mathbf{a} \cdot \mathbf{b} = |a||b| \cos\theta$.

Now put the acceleration you've calculated into your Euler's method integrator, and check that the ball does indeed travel in a circle.

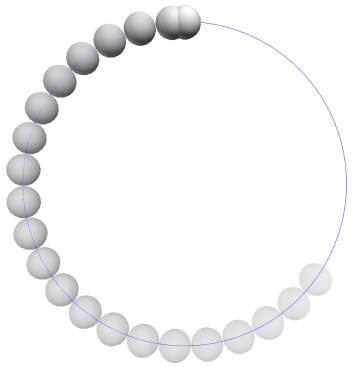
4.4 Magnetism, cross products and Lorentz force law

In the last problem, you solved for a force, given circular motion. How about the other way, what sorts of forces might lead to circular motion?

One interesting force that can lead to circular motion is the force a magnetic field exerts on a charged particle. This force is governed by the *Lorentz force law*

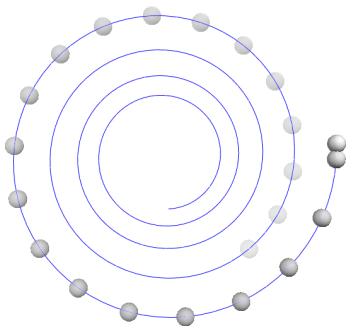
$$\mathbf{F} = q\mathbf{v} \times \mathbf{B} \tag{4.8}$$

Figure 4.3: Problem 4.4 Circular motion



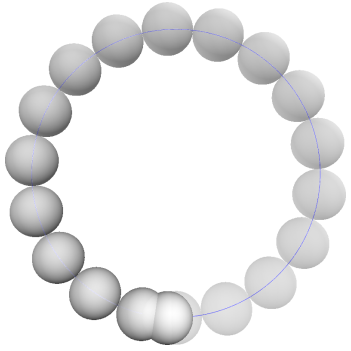
Recall that the *cross product* two vectors produces a third vector that is orthogonal (i.e. perpendicular) to each of them. This has can result in circular motion, since as we saw in Problem 4.4, circular motion involves a force at right angles to the velocity.

```
>>> B = vector(0,0,0.5e-4) # Tesla, the earth's magnetic field
>>> q = -1.6e-19 # Coulomb, the charge on an electron
>>> m = 9e-31 # kg, the mass of an electron
>>>
>>> s = sphere(radius=2.0e-6)
>>> s.velocity = vector(100,0,0) # m/s
>>> trail=curve(color=color.blue,pos=s.pos)
>>>
>>> t=0
>>> dt=1e-8 # s
>>> while t < 3e-6:
...   s.acceleration = q*cross(s.velocity,B)/m
...   s.velocity = s.velocity + s.acceleration*dt
...   s.pos = s.pos + s.velocity*dt
...   trail.append(s.pos)
...   t = t+dt
...   rate(4e-7/dt)
```

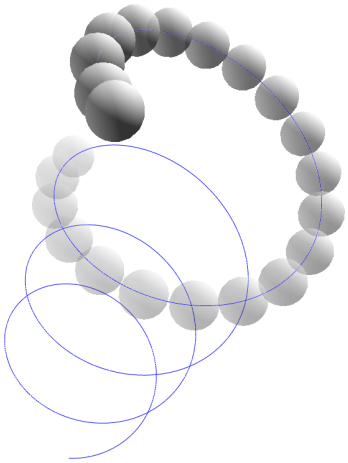


This doesn't quite look like circular motion. Something is obviously wrong here. The problem is that our value of Δt is too large. If we decrease Δt to a considerably smaller value, we get much better behavior.

```
>>> dt=1e-10 # s (use a smaller value for dt)
```



Note that although we have apparently solved this problem by decreasing the step size, in reality we have only reduced its magnitude. The ball is still



spiraling, just very slowly. A true solution will involve using the *Verlet* method (see Section 6.2) to eliminate this systematic error entirely.

```
>>> dt=1e-10 # s (use a smaller value for dt)
```

The Lorentz force law does not always lead to circular motion. With different initial conditions, one can observe helical motion, as shown in the margin.

Problem 4.5 The cyclotron frequency Using the Lorentz force law together with the relationship between centripetal force, speed and radius (see Problem 4.4), analytically compute the frequency of an electron in a uniform magnetic field, as discussed in Section 4.8.

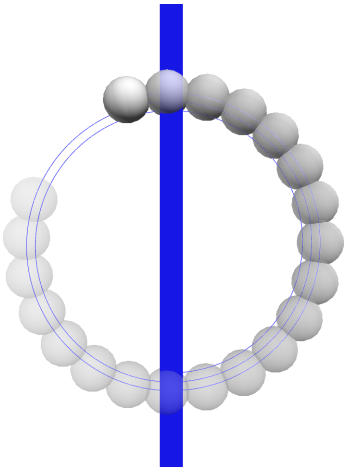
Work out the mass a particle with the electron's charge must have in order to have a cyclotron frequency of 1Hz. What velocity must this particle have in order for the radius of its motion to be 10m? Verify by numerical simulation that this answer is correct.

Problem 4.6 Modelling a cyclotron The cyclotron was an early form of particle accelerator that relied on placing charged particles in a confining magnetic field and introducing a periodic electric field to accelerate them. You will model a cyclotron with a very simple interaction. We will arrange that the electron feels a force

$$\mathbf{F} = 1.6 \times 10^{-22} \cos(\omega_c t)\hat{\mathbf{x}} \quad \text{Newtons} \tag{4.9}$$

whenever the $\hat{\mathbf{x}}$ component of its position has a magnitude less than 1 μm. Here ω_c is the *cyclotron frequency* computed in Problem 4.5. What happens if you use twice the cyclotron frequency? Some other value?

Figure 4.4: Problem 4.6 Modelling a cyclotron



5 Friction—it's always present

Friction and viscous forces are always present to some degree in any mechanical experiment[1], but it's seldom treated, largely because it complicates analytical computations. Fortunately, in numerical simulations viscous forces present no particular difficulty.

5.1 Air friction—viscous fluids

The behavior of the viscous force will depend on the velocity, size, shape and materials of an object, as well as the intrinsic properties of the fluid. This complexity is another reason viscous forces are often neglected: it's not a problem that lends itself to simple equations.

Fortunately, the viscous force on a *spherical* object moving *slowly* in any fluid[2] will be

$$\mathbf{F} = -m\gamma\mathbf{v} \tag{5.1}$$

where m is the mass, and I have defined a new constant γ, which clearly has units of s^{-1}. The parameter γ here can be thought of (crudely) as a decay rate for the speed. In fact, the solution to Newton's Second Law when viscous drag is the only force is simply

$$\mathbf{v}(t) == \mathbf{v}(0)e^{-\gamma t} \tag{5.2}$$

We can also connect γ with intuition by considering the question of terminal velocity when in free fall. We have reached terminal velocity when the velocity becomes constant, which means that the net force must be zero. The net force of an object under viscous drag in gravity is

$$\mathbf{F} = m\mathbf{g} - m\gamma\mathbf{v} \tag{5.3}$$

where $\mathbf{g}$ is the vector acceleration due to gravity. When we set the net force to zero, the velocity becomes the terminal velocity and we can solve for it.

$$\mathbf{v}_{terminal} = \frac{\mathbf{g}}{\gamma} \tag{5.4}$$

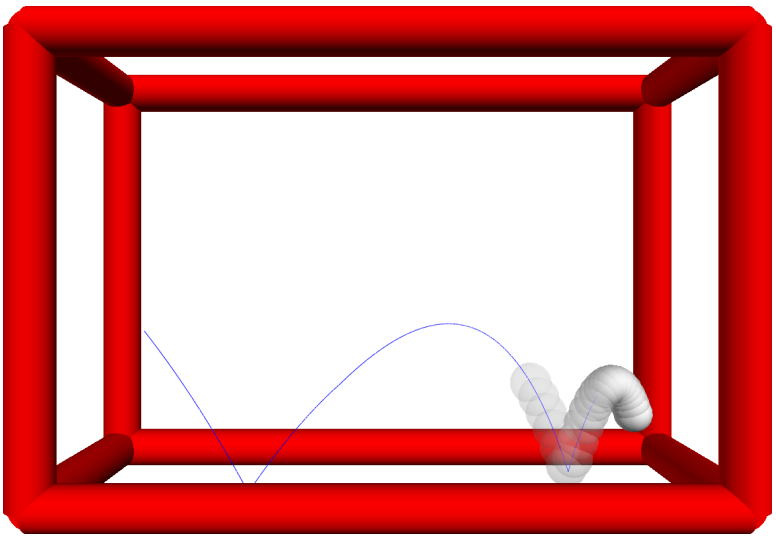
[1] We understand this as a result of the Second Law of Thermodynamics

[2] This is a universal property that can be understood through a consideration of the Taylor expansion of the viscous force, which for moderate speeds is a well-behaved function of the speed. For sufficiently small speed, the first—linear—term in the expansion must dominate.

from which we can see that the terminal speed is just g/γ.

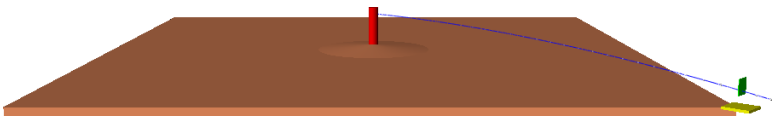
Problem 5.1 Ball in a viscous box Add viscous drag to the ball in a box with gravity from Problem 4.3.

Figure 5.1: Problem 5.1 Ball in a viscous box



Problem 5.2 Baseball II Add viscosity to the baseball problem, starting with your solution to Problem 4.1. Use Equation 5.1 with a value of γ corresponding to a terminal speed for the baseball of 50 m/s. Note that this is a poor approximation, as a fastball can hardly be described as moving slowly through the air.

Figure 5.2: Problem 5.2 Baseball II



5.2 Air drag at high speeds

The viscous force described in the previous section is correct for objects moving at low speeds, but what happens when objects start moving faster? Naturally, the higher powers in the speed will begin to dominate. For ordinary-

sized objects[3] in air—which isn't very viscous—the drag force is approximately quadratic.

$$\mathbf{F}_{drag} = -\frac{1}{2} C_d \rho A \left|\mathbf{v}\right| \mathbf{v} \tag{5.6}$$

where A is the cross-sectional area, ρ is the fluid density, and C_d is a unitless constant relating to the object's shape and surface properties, which we will take to be one.

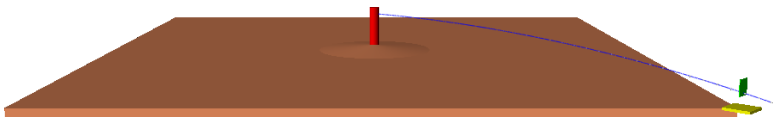
Given Equation 5.6, we can compute the terminal velocity of an object falling in air, as we did in Section 5.1. By setting the net force due to gravity plus the drag force to zero, we obtain

$$v_{terminal} = \sqrt{\frac{2mg}{C_d \rho A}} \tag{5.7}$$

Problem 5.3 Terminal velocity of a human Given that the density of air is 1.2 kg/m^3 and using $C_d \approx 1$, estimate your terminal speed in air. One mph is 0.447 m/s.

Problem 5.4 Baseball III Modify your solution to Problem 5.2 to use the more realistic drag force from Equation 5.6 which is quadratic in the speed of the baseball. The mass of a baseball is 145 g.

Figure 5.3: Problem 5.4 Baseball III



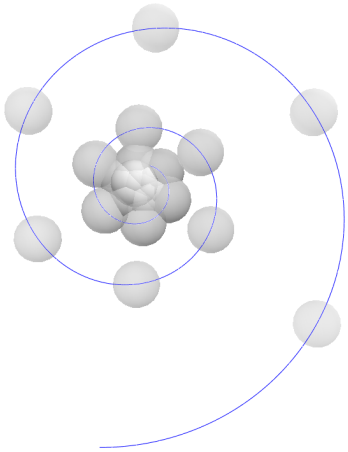
At what angle does the pitcher need to throw the ball in order to hit the strike zone?

Problem 5.5 Bremsstrahlung Below is an image from a bubble chamber, which shows the tracks left by electron-positron pairs that are created. The bubble chamber is placed in a magnetic field, and the electrons and positrons both lose energy continuously due to *bremsstrahlung* ("braking radiation"). Try modelling an electron in a magnetic field with a viscous force, and see if you can predict by comparing your results with those in the image whether the

[3] Technically, the distinction is in the *Reynolds number*, which is a measure of the effective viscosity in a fluid.

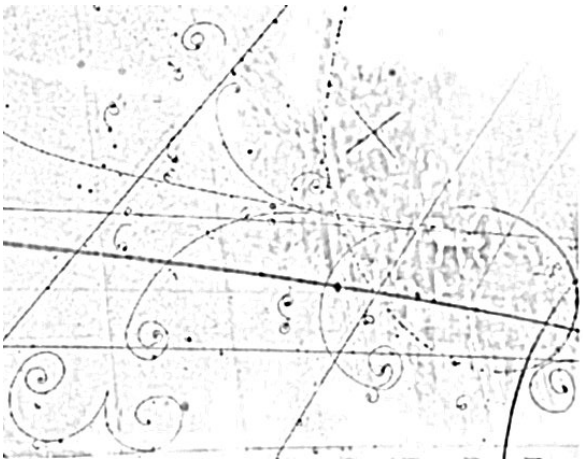
$$R = \frac{vL}{\nu} \tag{5.5}$$

where v is the speed at which an object is moving relative to the fluid, L is its linear dimension, and ν is the *kinematic viscosity*, which has units of m^2/s. The kinematic viscosity of water is 1×10^{-6}mm^2/s, while that of air is about 1.5×10^{-5} m^2/s. In practice The linear *viscous* drag from Section 5.1 is really more relevant for ordinary objects in water, or tiny objects (such as dust or insects) in air.



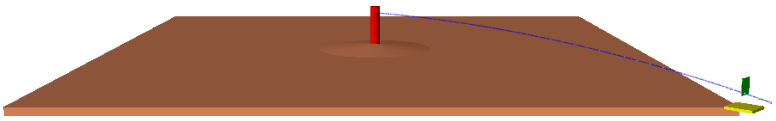
Problem 5.5 Bremsstrahlung

bremsstrahlung force is proportional to the speed or the square of the speed.



Problem 5.6 Wind Try changing your baseball solution from Problem 5.4 to include a cross-wind of 5 miles per hour. Does the pitch still hit the strike zone? What speed does the wind need to be in order for a pitch that starts out aimed directly at home plate to be a ball?

Figure 5.4: Problem 5.6 Wind



Problem 5.7 Thermostat Implement a *Langevin thermostat*, which is a technique in molecular dynamics for modeling objects in contact with a thermal reservoir. The Langevin thermostat involves the introduction of a fictitious fluid that weakly interacts with your objects. The result is that there is a viscous drag, as well as some *Brownian motion*. The interaction takes the form of a force such as

$$\mathbf{F}_{Langevin} = \sqrt{\frac{2\gamma m k_B T}{\Delta t}}\,\mathbf{R}(t) - \gamma m \mathbf{v} \tag{5.8}$$

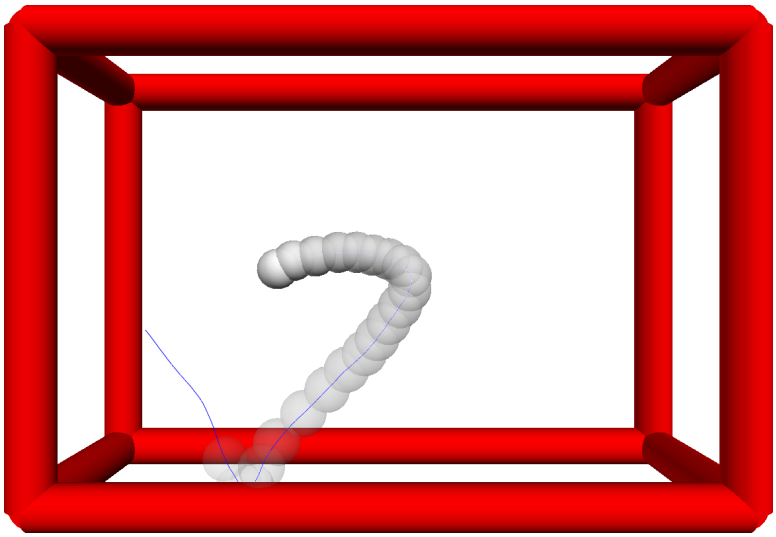
where T is the temperature, k_B is Boltzmann's constant, γ is a parameter defining the strength of the thermostat interaction, and $\mathbf{R}(t)$ is a vector that changes direction every Δt with a random gaussian (normal) distribution and variance one. This is an unusual situation, where the magnitude of the random force depends on Δt, which is an arbitrarily chosen small time step. This is actually correct because of the properties of a random walk.

In python, you can obtain a random number with normal distribution, zero mean and unit variance by using

```
>>> import random
>>> print(random.gauss(0,1))
>>> def randomVector():
...   return (vector(random.gauss(0,1),
...                  random.gauss(0,1),
...                  random.gauss(0,1)))
```

Implement this thermostat by modifying your ball-in-a-box from Problem 5.1. How high does the temperature need to be set to cause the ball to reach the top of the box?

Figure 5.5: Problem 5.7 Thermostat



6 Energy conservation

Conservation of energy is a fundamental law of physics that states that the total amount of energy in an isolated system remains constant. In other words, if one sort of energy is lost, the same amount of energy must have been gained elsewhere. The simplest sort of energy is *kinetic energy*,[1] which can be defined for a point particle (or non-rotating rigid object) as

$$T = \frac{1}{2}mv^2 \tag{6.1}$$

It should be immediately clear that the kinetic energy alone is not conserved, since a ball that is dropped begins with no kinetic energy, and later gains kinetic energy. Energy conservation is obeyed, however, because the ball is losing *potential energy*. The potential energy V due to a uniform gravitational field $\mathbf{g}$ is given by

$$V = -m\mathbf{g} \cdot \mathbf{r} + \text{constant} \tag{6.2}$$

The potential energy is only defined up to an arbitrary constant. This physically relates to its derivation as an integral of the force.

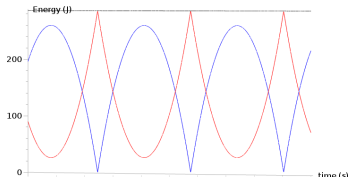
6.1 Energy conservation in a gravitational field

Consider the ball in a box with gravity solved in Problem 4.3. Let's check that energy is conserved in this problem.

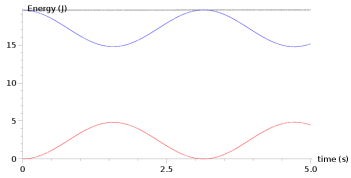
```
>>> from visual.graph import * # import graphing features
>>>
>>> gdisplay(xtitle='time (s)', ytitle='Energy (J)',
...          foreground=color.black, background=color.white)
>>> energy = gcurve()
>>> potential = gcurve(color=color.blue)
>>> kinetic = gcurve(color=color.red)
>>>
>>> # skpping various initializations
>>> m = 2 # kg
>>> g = vector(0,-9.8,0) # m/s^2
```

[1] The conventional symbol for kinetic energy is T.

```
>>> while t < 10:
...     # ...
...     # skipping solution to earlier homework problem
...     # ...
...     kineticEnergy = 0.5*m*dot(s.velocity,s.velocity)
...     potentialEnergy = -m*dot(g,s.pos+ymax*vector(0,1,0))
...     kinetic.plot(pos=(t,kineticEnergy))
...     potential.plot(pos=(t,potentialEnergy))
...     energy.plot(pos=(t,kineticEnergy + potentialEnergy))
...     t = t+dt
...     rate(1/dt)
```



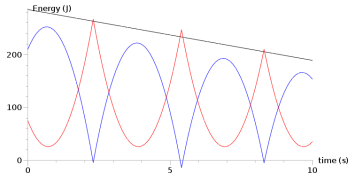
Problem 6.1 Energy conservation in a magnetic field Plot the energy as a function of time for a 0.1 kg object with a charge of 0.2 C (which is a *huge* charge) moving in a magnetic field of 1 T (which is a *huge* magnetic field) under the influence of gravity.




```
>>> B = vector(0,0,1) # Tesla
>>> q = 0.2 # Coulomb
>>> m = 0.1 # kg
>>> g = 9.8*vector(0,-1,0) # m/s**2
```

Play with different initial velocities and positions. Is energy conserved in this system? What kinds of trajectories can you create?

6.2 Verlet's method



We are forced to use a very small Δt in order to achieve energy conservation in our calculations. Failure to use an adequately small Δt when using Euler's method leads to a lack of energy conservation, as illustrated in the margin. This is a failing of Euler's method, and relates to our use of a *forward* derivative in deriving Euler's method. A nicer approach is *Verlet's method*, which involves the use of centered derivatives only by directly integrating the second-order differential equation that is Newton's second law.

$$m\mathbf{a} = \mathbf{F} \tag{6.3}$$

$$\frac{d^2\mathbf{r}}{dt^2} = \frac{\mathbf{F}}{m} \tag{6.4}$$

By applying twice the centered definition of a first derivative (see Equation 3.4), we obtain the following description of the second derivative

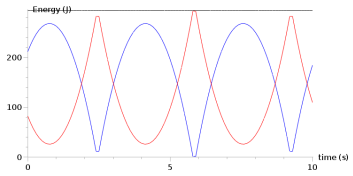
$$\frac{d^2\mathbf{r}}{dt^2} = \lim_{\Delta t \to 0} \frac{\mathbf{r}(t+\Delta t) + \mathbf{r}(t-\Delta t) - 2\mathbf{r}(t)}{\Delta t^2} \tag{6.5}$$

If we set this equal to $\mathbf{F}/m$ and solve for $\mathbf{r}(t+\Delta t)$, we arrive at

$$\mathbf{r}(t+\Delta t) = 2\mathbf{r}(t) - \mathbf{r}(t-\Delta t) + \frac{\Delta t^2}{m}\mathbf{F} \tag{6.6}$$

To begin, let's implement our ball in a box with gravity using Verlet's method.

```
>>> dt = 0.1  # seconds
>>> s.oldpos = vector(0,0,0) # m
>>> s.pos = s.oldpos + vector(5,8,1)*dt
>>> m = 2 # kg
>>> g = vector(0,-9.8,0) # m/s^2
>>> while t < 10:
...   # the verlet integration:
...   s.force = m*g
...   olderpos = s.oldpos
...   s.oldpos = 1*s.pos
...   s.pos = 2*s.pos - olderpos + s.force*dt**2/m
...   # work out the energy using a centered derivative
...   # for the velocity:
...   velocity = (s.pos - olderpos)/(2*dt)
...   kineticEnergy = 0.5*m*dot(velocity,velocity)
...   potentialEnergy = -m*dot(g,s.oldpos+ymax*vector(0,1,0))
...   print(kineticEnergy + potentialEnergy)
...   # Now let's keep the ball in the box:
...   if s.pos.x > xmax or s.pos.x < -xmax:
...     s.pos.x, s.oldpos.x = s.oldpos.x, s.pos.x
...   if s.pos.y > ymax or s.pos.y < -ymax:
...     s.pos.y, s.oldpos.y = s.oldpos.y, s.pos.y
...   if s.pos.z > zmax or s.pos.z < -zmax:
...     s.pos.z, s.oldpos.z = s.oldpos.z, s.pos.z
...   t = t+dt
...   rate(1/dt)
```



As you can see, we now have extremely accurate energy conservation, even though we are using an extremely large value for Δt. This is one of the primary reasons why Verlet is so very popular among computational physicists, and is a result of the use of the unbiased centered derivative in its derivation.

6.3 Energy conservation in presence of friction

There are more forms of energy than kinetic and potential energy. One of the most common forms is *thermal energy*. Thermal energy is composed of kinetic and potential energy on an atomic scale. There is not much we will say in this course about thermal energy, except that it is usually where energy that is "lost" due to friction ends up. Given that we know so little about thermal energy, what can we do to ensure that our energy remains conserved? The key

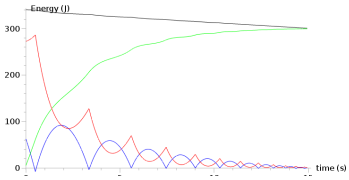
is in the definition of *work*. Work is a measure of the transfer of energy, and in general is defined by

$$W = \int \mathbf{F} \cdot \mathbf{dr} \tag{6.7}$$

which tells us that in order to determine the work, if we know the force and $\mathbf{r}(t)$, we must perform a *line integral*. Line integrals are not particularly scary, but we can make this even simpler if we consider the *power*, which is the rate at which work is done.

$$P = \mathbf{F} \cdot \frac{d\mathbf{r}}{dt} \tag{6.8}$$

$$= \mathbf{F} \cdot \mathbf{v} \tag{6.9}$$



Problem 6.2 Energy of a ball in a viscous box

So for a simple linear viscosity as described in , we can now work out the energy lost to heat in a time Δt, which is just

$$\Delta W = P\Delta t \tag{6.10}$$

$$= \mathbf{F} \cdot \mathbf{v}\Delta t \tag{6.11}$$

Problem 6.2 Energy of a ball in a viscous box Check the energy conservation of your solution to Problem 5.1, taking into account the energy lost to friction. Try this check with a range of values for Δt. How small does Δt need to be to give reasonable energy conservation? Is the answer to this question affected by how long you run the simulation?

Problem 6.3 Verlet with viscosity Derive (on paper!) an update equation using Verlet's method for a ball in a viscous fluid that is subjected to an arbitrary external force $\vec{F}_{\text{external}}(t)$. You will need to write the analogue of Equation 6.6 including a viscous force explicitly as a centered derivative as well as $\vec{F}_{\text{external}}(t)$. Keep in mind that the viscous force is

$$\vec{F}_{\text{viscous}}(t) = -m\gamma\vec{v}(t) \tag{6.12}$$

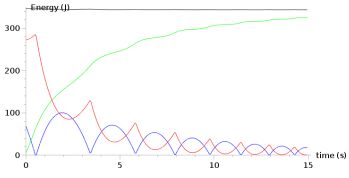
Recall that the centered formula for a second derivative is

$$\frac{d^2y}{dx^2} = \lim_{\Delta x \to 0} \frac{y(x+\Delta x) + y(x-\Delta x) - 2y(x)}{\Delta x^2} \tag{6.13}$$

Show your work!
Note: $\vec{v}$ should not appear in your final answer

Problem 6.4 Ball in a viscous box using Verlet's method Solve Problem 5.1 using Verlet's method instead of Euler's method. In order to deal with the viscous drag term, you will need the results of Problem 6.3, which is the analogue of Equation 6.6 including the viscous drag term explicitly as a centered derivative.

Check that energy is now conserved if you take into account the energy lost to friction. Try this check with a wide range of values for Δt. How well is energy conserved, even for values of Δt that are not small, or long running times?



Problem 6.4 Ball in a viscous box using Verlet's method

7 Hooke's law—springs

7.1 Hooke's law

Hooke's law states that the restoring force exerted by a spring is proportional to its displacement from equilibrium.

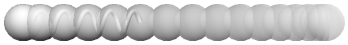
$$\mathbf{F} = -k\Delta\mathbf{r} \tag{7.1}$$

where the constant k is the *spring constant*. Hooke's law actually applies to considerably more than just mechanical springs, because any system with a stable equilibrium will respond to small deviations according to Hooke's law—although in general there may be different spring constants in different directions.

7.2 Motion of a spring

Let us attach a sphere to the origin with a simple spring. The code will be identical to code we've seen many times, only with a new law for the force.

```
>>> m = 2 # kg
>>> k = 4 # N/m
>>>
>>> t=0
>>> dt=0.01 # s
>>>
>>> s=sphere()
>>> spring = helix(radius=0.6, thickness=0.3)
>>> s.pos = vector(10,0,0) # m
>>> s.oldpos = s.pos - vector(0,0,0)*dt # m
>>>
>>> while t < 0.5*pi:
...   s.force = -k*s.pos
...   s.pos, s.oldpos = \
...          2*s.pos - s.oldpos + s.force*dt*dt/m, 1*s.pos
...   spring.axis = s.pos -  spring.pos
...   t = t+dt
...   rate(1/dt)
```



As you can see, the motion is sinusoidal. Having observed this, we can compute the frequency by considering Newton's second law

$$\mathbf{F} = m\mathbf{a}, \quad \text{and from Hooke's law} \tag{7.2}$$

$$-k\mathbf{r} = m\frac{d^2\mathbf{r}}{dt^2} \tag{7.3}$$

If we insert $\mathbf{r}(t) = \mathbf{r}(0)\cos(\omega t)$ into the above equation, we obtain

$$-k\mathbf{r}(0)\cos(\omega t) = -m\omega^2\mathbf{r}(0)\cos(\omega t) \tag{7.4}$$

$$\omega = \sqrt{\frac{k}{m}} \tag{7.5}$$

7.3 Energy in a spring

The energy stored in a spring can be computed in the same way that we computed the energy lost to heat, by integrating the force to find the work. However, because the force due to a spring is a function of position *only*, we can do this analytically.

$$W = \int \mathbf{F}\cdot\mathbf{dr} \tag{7.6}$$

$$= -k\int \mathbf{r}\cdot\mathbf{dr} \tag{7.7}$$

$$= -\frac{1}{2}k\mathbf{r}\cdot\mathbf{r} \tag{7.8}$$

$$= -\frac{1}{2}kr^2 \tag{7.9}$$

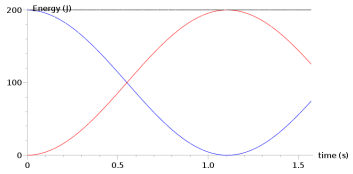
We've got a little confusion here, because of the ambiguity between the work done *by* the spring and the work done *on* the spring, which are opposite in sign. Nevertheless, the potential energy stored in a spring is quite easy to compute, and comes out to simply

$$V = \frac{1}{2}kr^2 \tag{7.10}$$

We can now write a quick program to verify the conservation of energy in our spring:

```
>>> while t < 0.5*pi:
...   s.force = -k*s.pos
...   olderpos = s.oldpos # save oldpos for the the velocity
...   s.pos, s.oldpos = \
...       2*s.pos-s.oldpos+s.force*dt*dt/m, 1*s.pos
...   velocity = (s.pos - olderpos)/(2*dt)
...   kineticEnergy = 0.5*m*dot(velocity,velocity)
...   potentialEnergy = 0.5*k*dot(s.oldpos,s.oldpos)
...   kinetic.plot(pos=(t,kineticEnergy))
```

```
...    potential.plot(pos=(t,potentialEnergy))
...    energy.plot(pos=(t,kineticEnergy + potentialEnergy))
```



As you can see, the energy alternates sinusoidally between kinetic and potential energy.

Problem 7.1 Damped oscillations Simulate a ball on a spring, moving with a viscous damping term γ, keeping the mass at 2 kg and the spring constant at 4 N/m. Solve this problem using Verlet's method, as in Problem 6.4, and plot the kinetic, potential, thermal and total energy versus time for the damped oscillator as you did in that problem. By changing the value γ you can make the spring either *over damped*, *under damped* or *critically damped*. Experiment a bit and see if you can find these regimes. What values of γ do they correspond to?

Figure 7.1: Problem 7.1 Damped oscillations

Would you like the shock absorbers in your car to be over damped, under damped or critically damped?

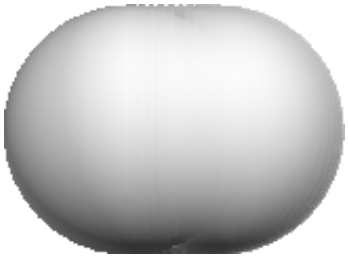
Problem 7.2 Driven, damped oscillations Implement a *driven* damped oscillator by adding to the solution to the previous problem an additional external force

$$\mathbf{F} = \mathbf{f}_{max} \sin(\omega t) \tag{7.11}$$

where you may take f_{max} to be 2 Newtons. Be sure to run the simulation for at least ten periods (or at least $1/\gamma$, if γ is small). The motion should eventually fall into a steady state.

How does the motion change when you change the driving frequency ω? What is the effect of changing the damping γ? The frequency that leads to the maximum amplitude of motion is termed the *resonant frequency*.

Figure 7.2: Problem 7.2 Driven, damped oscillations



For extra fun: Plot the kinetic, potential, thermal and total energy versus time for the driven damped oscillator. What happens to the total energy? Why?

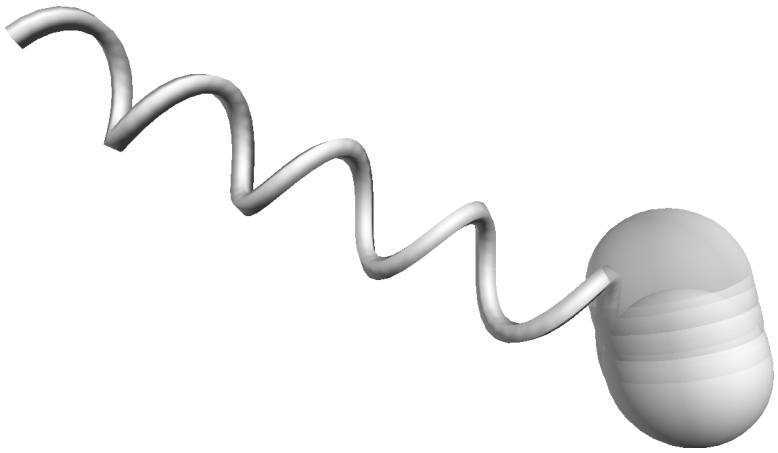
For extra fun: Use the Verlet method.

Problem 7.3 Spring pendulum Model a real spring with equilibrium length $L = 10$ m, under the influence of gravity. The spring's *equilibrium length* is defined as the length at which the force exerted by the spring is zero. Determining the magnitude and direction of the force will be a bit trickier in this case, since the formulas in the text only describe springs with an equilibrium length of zero. Solve this problem both using the Verlet method, and using Euler's method.

Plot each component of the energy—keeping separate account of gravitational and spring potential energy—as well as the total energy. How would you set the initial conditions such that the potential energy stored in the spring is constant?

How does the behavior of this spring pendulum change when you increase the spring constant k? If you have extra time, try adding in damping and/or a periodic driving force.

Figure 7.3: Problem 7.3 Spring pendulum



8 Newton's third law

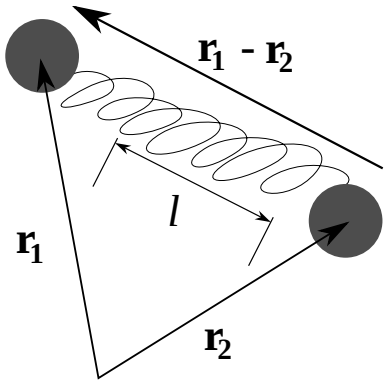
So far, we've only considered a single object, which is quite a simplification. Once we have multiple objects, we need to consider forces that they exert on each other. *Newton's third law* states that if one object exerts a force on another, the second object must exert an equal and opposite force on the first. This law is really all we need in order to start working on multiple objects.

8.1 Spring dumbell

Let us consider the problem of two balls connected by a spring with an equilibrium length l. The force on one ball will is given by Hooke's law, which now looks a little different since our spring has a non-zero equilibrium length:

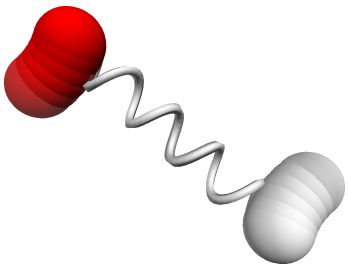
$$\mathbf{F}_1 = -k(|\mathbf{r}_1 - \mathbf{r}_2| - l)\frac{\mathbf{r}_1 - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_2|} \tag{8.1}$$

By Newton's third law, the force the second ball will be equal in magnitude and opposite in the direction of the force on the first ball.




```
>>> L = 10 # m
>>> m = 2 # kg
>>> k = 4 # N/m
>>>
```

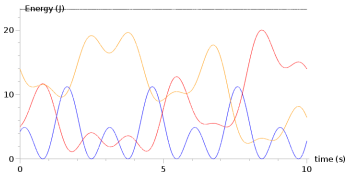
```
>>> t=0
>>> dt=0.001 # s
>>>
>>> s1=sphere(pos=vector(8,-3,0)) # m
>>> s2=sphere(pos=vector(0,0,0), color=color.red)
>>> spring = helix(radius=0.6, thickness=0.3)
>>> s1.oldpos = s1.pos - dt*vector(-3,-2,1)
>>> s2.oldpos = s2.pos - dt*vector(3,2,1)
>>>
>>> while t < 6:
...   dr = s1.pos - s2.pos
...   s1.force = -k*(abs(dr) - L)*dr/abs(dr)
...   s2.force = -s1.force
...   s1.pos, s1.oldpos = \
...       2*s1.pos - s1.oldpos + s1.force*dt*dt/m, 1*s1.pos
...   s2.pos, s2.oldpos = \
...       2*s2.pos - s2.oldpos + s2.force*dt*dt/m, 1*s2.pos
...   spring.pos = s2.pos
...   spring.axis = s1.pos - s2.pos
...   t = t+dt
...   rate(1/dt)
```



Problem 8.1 Energy conservation with two particles The potential energy stored in the spring is determined by the distance between the two balls

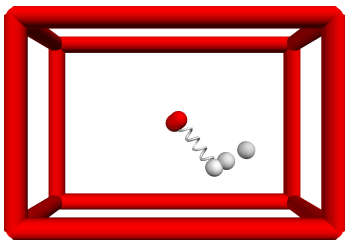
$$V = \frac{1}{2}k\left(|\mathbf{r}_1 - \mathbf{r}_r| - L\right)^2 \tag{8.2}$$

and now we have two kinetic energies to compute. Verify energy conservation in this system, plotting separately each of the two kinetic energies. You will observe that if the velocities of the two balls are not equal and opposite, the kinetic energy has a somewhat irregular behavior.



Problem 8.1 Energy conservation with two particles

Problem 8.2 Dumbell in a box Put the two masses connected by a spring which we modelled in Section 8.1 into a box with hard walls, and give the balls an initial velocity so that they bounce off the walls. Then check that energy is conserved in this system.



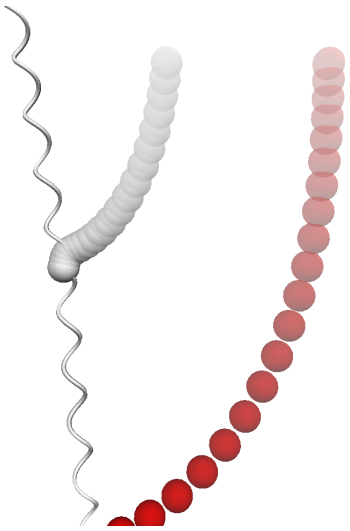
Problem 8.2 Dumbell in a box

Problem 8.3 Double pendulum Consider the problem of two 2 kg masses forming a chain held together by a spring in the presence of gravity, but with one of the two masses attached by a second spring to a fixed point as illustrated in the accompanying figure. Model this double pendulum using the parameters defined below.

```
>>> L = 10 # m
>>> m = 2 # kg
```

```
>>> k = 10 # N/m
>>> g = vector(0,-9.8,0) # m/s^2
```

Can you find initial conditions that lead to periodic motion? Bonus: check that conservation of energy for your solution.



Problem 8.3 Double pendulum

8.2 Momentum conservation

Newton's third law tells us that when there are two objects that are interacting only with each other, the force on one must be exactly opposite to the force on the other. Newton's second law informs us that

$$\mathbf{F} = m\frac{d\mathbf{v}}{dt} \tag{8.3}$$

which we can combine with Newton's third law, to obtain a relation between the velocities of the two objects:

$$\mathbf{F_1} = -\mathbf{F_2} \tag{8.4}$$

$$m_1\frac{d\mathbf{v_1}}{dt} = -m_2\frac{d\mathbf{v_2}}{dt} \tag{8.5}$$

We can then move everything to one side, to obtain

$$\frac{d}{dt}\left(m_1\mathbf{v_1} + m_2\mathbf{v_2}\right) = 0 \tag{8.6}$$

which tells us that the combination of quantities $m_1\mathbf{v_1} + m_2\mathbf{v_2}$ is independent of time—i.e. is conserved. This quantity is what we call the *total momentum* $\mathbf{p}_{tot}$. The momentum of just one object is $m_1\mathbf{v_1}$.[1] Momentum is conventionally written as $\mathbf{p}$.

Problem 8.4 Collisions A commonly used approximation to the repulsive interaction between two atoms is

$$\mathbf{F}_1 = k\frac{\mathbf{r}_1 - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_2|^{13}} \tag{8.7}$$

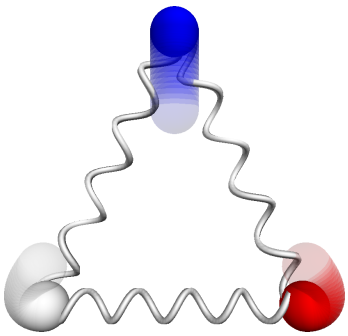
Implement this interaction for two balls with the following parameters, and check whether the total momentum is indeed independent of time.

```
>>> m1 = 2 # kg
>>> m2 = 1 # kg
>>> k = 10 # N/m^12
>>>
...   # see whether the total momentum changes
...   print("Momentum is:" + str(momentum1 + momentum2))
```

Figure 8.1: Problem 8.4 Collisions

Try using different masses and different initial velocities. Does momentum remain conserved? What happens if you increase the time step Δt to an unreasonably large value?

Problem 8.5 Normal modes—coupled springs Now let's consider three balls connected by springs, as shown in the figure, where l given below is the equilibrium length of each spring, and k is the spring constant. Let the initial velocity of each ball be zero, and try using initial positions such that the three balls are *not* each separated by a distance l.

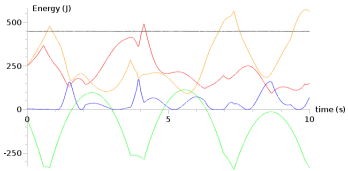


Problem 8.5 Normal modes—coupled springs

```
>>> L = 10 # m
>>> m = 2 # kg
>>> k = 1 # N/m
...   # see whether the total momentum changes
...   print("Momentum is:" + str(momentum1 + momentum2 + momentum3))
```

Is momentum conserved in this system? Does your answer change if you give the balls some asymmetrical initial velocity?

What initial positions can you find that lead to simple periodic motion? This different motions—which have a distinct frequency—is called a *normal mode*. How many normal modes can you find? It may help to consider the symmetry of this problem.



Problem 8.6 Dumbell in a box with gravity

Problem 8.6 Dumbell in a box with gravity Let's add gravity to the dumbell in a box we modelled in Problem 8.2. Now check for conservation of both energy and momentum. Which is conserved? Why or why not?

[1] Special relativity changes our understanding of momentum somewhat. It remains true that $\mathbf{F} = \frac{d\mathbf{p}}{dt}$, but we can no longer simply state that $\mathbf{p} = m\mathbf{v}$. In reality, the momentum becomes infinite as an object's speed approaches the speed of light, with the exception being a particle with zero rest mass, which must always travel at the speed of light.

9 Inverse square law—gravity

So far, we have assumed that the gravitational force is a constant $m\mathbf{g}$. Near the surface of the earth, this is quite a good approximation, but it is indeed only an approximation. The gravitational force between two spherical bodies[1] is actually

$$\mathbf{F} = Gm_1m_2\frac{\mathbf{r}_1 - \mathbf{r_2}}{|\mathbf{r}_1 - \mathbf{r_2}|^3} \tag{9.1}$$

where the gravitational constant G is 6.67300×10^{-11} Nm^2/kg^2. You may think at first glance that this doesn't look much like an *inverse square law* since the exponent in the denominator is three. However, the *magnitude* of the force goes as the inverse of the square of the distance between the two objects.

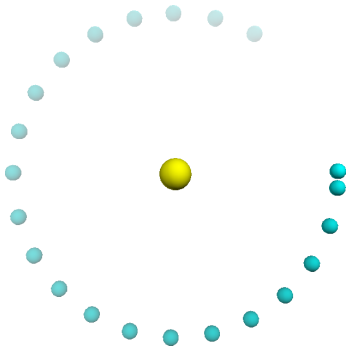
Problem 9.1 Length of the year This is a paper and pencil problem. The earth is 149.6×10^9 m from the sun. The mass of the sun is 1.99×10^{30} kg and the mass of the earth is 5.97×10^{24} kg. From this data, assuming the earth moves in a circular orbit, work out the length of the year using centripetal acceleration from the results of Problem 4.4. What is the speed of the earth?

9.1 Planetary motion

Let's now simulate the motion of the earth around the sun. We'll use the verlet approach.

```
>>> m_e = 5.97e24 # kg
>>> m_s = 1.99e30 # kg
>>> G = 6.67300e-11 # Nm^2/kg^2
>>> au = 149.6e9 # m
>>> year = 31556926 # s
>>>
>>> sun=sphere(radius = 0.1*au)
>>> earth=sphere(radius = 0.05*au)
```

[1] This formula is also accurate for non-spherical bodies, provided their dimensions are much smaller than the distance between the two bodies. Corrections to this equation are generally called "tidal forces," and relate to the effect that causes the tides, and caused the moon to rotate in sync with its revolution around the earth so we always see the same side of the moon.



```
>>> #earth.v0 = ???  m/s
>>> earth.r = vector(au,0,0)
>>> earth.rold = earth.r - dt*earth.v0
>>> sun.r = vector(0,0,0)
>>> sun.rold = sun.r
>>>
>>> while t <= year:
...   r = sun.r - earth.r
...   f = G*m_s*m_e*r/abs(r)**3
...   earth.force = f
...   sun.force = -f
...   earth.r, earth.rold = \
...       2*earth.r-earth.rold+earth.force*dt*dt/m_e, earth.r
...   sun.r, sun.rold = \
...       2*sun.r-sun.rold+sun.force*dt*dt/m_s, sun.r
...   earth.pos = earth.r
...   sun.pos = sun.r
...   t = t+dt
```

9.2 Gravitational energy

In Section 6 we used a potential energy of

$$V = -m\mathbf{g} \cdot \mathbf{r} \tag{9.2}$$

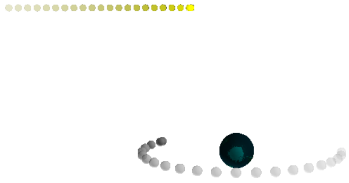
for gravity, which is a good approximation for positions close to the Earth's surface. How does this change when the gravitational force is taken as an inverse square law? We can obtain the answer by integrating the force over distance to find the work that it does. When we do this, we find that the potential energy of two spherical bodies is

$$V = -\frac{Gm_1m_2}{|\mathbf{r}_1 - \mathbf{r_2}|} \tag{9.3}$$

Problem 9.2 Energy conservation VI Check that energy is conserved when the Earth travels around the Sun. Perturb the Earth's orbit, and verify that energy is still conserved.

Problem 9.3 Throw in the moon Add the moon to the simulation of the solar system. Look up the relevant quantities on wikipedia or elsewhere on the web. Try making the spheres for the sun, moon and earth be to scale. Doing so will most likely require that you place the camera near the earth using code such as:

```
...   scene.center = earth.r
```



Problem 9.3 Throw in the moon

A Navigating in the unix shell

This is a short appendix on the subject of getting around the bash shell, which is used on most unix-like operating systems. Learning to use the command-line interface can be a rewarding and addictive experience. This introduction is organized so that you can read through it sequentially, working through the examples as you go, and gain a good working knowledge of the bash shell. Sections are also titled in order to make it this section useful as a reference.

A.1 echo

The simplest unix command is `echo`, which simply repeats its arguments.

```
$ echo Hello world.
Hello world.
```

This doesn't sound very interesting just yet, but you will find it suprisingly useful, particularly when you find some other command behaving in a suprising way. For instance, try running

```
$ echo ~
/home/droundy
```

What happened? It seems like `echo` is not behaving as advertised, but what really happened is that the shell itself translated the '`~`' into the location of your home directory. Your home directory is the directory you are in when you first log into your computer.

A.2 pwd *(Print Working Directory)*

`pwd` displays the current working directory:

```
$ pwd
/home/droundy
```

On POSIX (unix-like) systems—which includes MacOS X—directories (also known as folders) are separated by the '`/`' character rather than the '`\`' character used on Windows systems.

I am currently in the directory `/home/droundy`, which is my *home directory*, as we expect from our previous discussion. But how do we change directories?

A.3 cd *(Change Directory)*

You can change directories with the `cd` command, you simply type `cd` followed by the name of the directory you wish to go to. For instance, I can move to the parent directory of my home directory:

```
$ cd /home
$ pwd
/home
```

You can return to your home directory using the '~' character. To move into the parent of a given directory, you can use the special directory name '..'.

```
$ cd ~
$ pwd
/home/droundy
$ cd ..
$ pwd
/home
$ cd droundy
$ pwd
/home/droundy
```

When a *path* (directory location) starts with a '/' character, it is an *absolute* path, relative to the '/' (*root*) directory. If you specify any other sort of path, it is interpreted relative to the current directory.

We've now done a bit of navigation, but it's hard to know what other directories we can `cd` into.

A.4 ls *(LiSt)*

With the `ls` command, we can list the contents of any directory we can access. To start, let's run `ls` with no arguments.

```
$ ls
darcs    haskell  MyLibrary.bib  ph265  ph632     proposals  xmonad
Desktop  Music    papers         ph631  Pictures  Videos
```

Your home directory will probably be less cluttered than mine,[1] and quite likely won't have any interesting directories to explore. For something more interesting, let's ask what there is in `/home`

```
$ ls /home
droundy  monica
```

As you can see, only my wife and I have accounts on the machine I'm using. Since it's my work laptop, her home directory doesn't have anything interesting, so let's explore the root directory.

[1] To be honest, I cleaned mine up for this example...

```
$ ls /
bin   cdrom  etc   lib         media  opt   root  srv  tmp  var
boot  dev    home  lost+found  mnt    proc  sbin  sys  usr  vmlinuz
```

Most of these directories aren't very interesting, but let's take a look at one of them anyhow.

```
$ ls /bin
bash   cpio        echo      kill    more        pwd    su      which
cat    date        false     ln      mount       rm     tar     zcat
chgrp  df          grep      login   mountpoint  rmdir  touch   zgrep
chmod  dir         gunzip    ls      mv          sed    true    zless
chown  dmesg       gzip      mkdir   ping        sh     umount
cp     domainname  hostname  mktemp  ps          sleep  uname
```

This is where many of the programs we are using reside. You can see `echo`, `pwd` and `ls`. We do not see, however, `cd`, which is because it is actually a bash internal command, rather than an actual program.

A.5 mkdir *(MaKe DIRectory)*

Let's start making a few changes. We'll begin my creating a new directory in our home directory, in which to play.

```
$ mkdir learning-unix
```

At this point, let me share with you a cool bash feature. Suppose we want to `cd` into the `learning-unix` directory we just created. We don't need to type the entire name of the directory, just enough for bash to recognize which one we're talking about. We do this using what is called *command completion*. Try typing

```
$ cd l<TAB>
```

where by `<TAB>` I mean to press the tab key. The shell will immediately fill in the rest of the directory name. Or, if you have two directories that start with *'l'*, it will beep, to let you know that it couldn't find an exact match.

A.6 > *(create a file)*

You can create a file using the editor `kate`, but that editor is not universally available on unix-like systems, so in this appendix I'll show you another way to create a file. This is by *redirecting* the output of any command.

```
$ echo Hello world. > new-file
```

The `>` operator redirects the output of any command to whatever file is specified. If this file already exists, it overwrites it.

A.7 less *(view contents of a text file)*

There are several ways to view the contents of a text file, but one of the most versatile is the `less` command.

```
$ less new-file
Hello world.
$ ls
new-file
```

So we now see that the `>` operator did indeed behave as advertized.

A.8 mv *(MoVe, or rename a file)*

We can rename a file using the `mv` command, which also is used for moving a file into a different directory.

```
$ mv new-file newer-file
$ ls
newer-file
$ mkdir new-directory
$ ls
new-directory  newer-file
$ mv *-file new-directory/
$ ls
new-directory/
$ ls new-directory
newer-file
```

A.9 rm *(ReMove file)*

We're basically done, so let's clean up the directories we've made.

```
$ rm new-directory/newer-file
$ cd ..
$ rm learning-unix
rm: cannot remove ‘learning-unix’: Is a directory
$ rm -r learning-unix
```

In the last case, we use the "recursive" flag, `-r`, which instructs `rm` to remove both the directory and anything in it.

B Programming practice problems

This is a set of problems to be used for programming practice. These problems are intended to be worked by pairs of students without the use of a computer. The first student (the *programmer*) writes on paper a function (named `foobar`) that solves the problem. The second part of the exercise is for a second student (the *computer*) who should read program *without knowing what it is intended to do* and execute it, again on paper. Both skills—writing *and* reading programs—are important aspects of programming.

Programming on paper 1

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: Is the number even? Call your function `foobar(n)`.

Running a program on paper 1

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(1)`? _____
What is `foobar(2)`? _____
What is `foobar(8)`? _____
What is `foobar(9)`? _____
What is `foobar(15)`? _____
What is `foobar(___)` for a `n` of your choice? _____

Programming on paper 2

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: Is the number divisible by either 3 or 5? Call your function `foobar(n)`.

Running a program on paper 2

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(0)`? _____
What is `foobar(2)`? _____
What is `foobar(8)`? _____
What is `foobar(9)`? _____
What is `foobar(15)`? _____
What is `foobar(___)` for a `n` of your choice? _____

Programming on paper 3

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: What is the sum of all positive integers less than n? Call your function `foobar(n)`.

Running a program on paper 3

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(0)`? _____
What is `foobar(2)`? _____
What is `foobar(8)`? _____
What is `foobar(9)`? _____
What is `foobar(15)`? _____
What is `foobar(___)` for a `n` of your choice? _____

Programming on paper 4

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: What is the largest perfect square that is less than `n`? Call your function `foobar(n)`.

Running a program on paper 4

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(2)`? _____
What is `foobar(8)`? _____
What is `foobar(9)`? _____
What is `foobar(15)`? _____
What is `foobar(17)`? _____
What is `foobar(___)` for a `n` of your choice? _____

Programming on paper 5

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: What is the sum of the squares of all non-negative odd integers less than n? Call your function `foobar(n)`.

Running a program on paper 5

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(0)`? _____
What is `foobar(1)`? _____
What is `foobar(2)`? _____
What is `foobar(8)`? _____
What is `foobar(___)` for a `n` of your choice? _____

Programming on paper 6

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: What is the square root of n, to a precision of 0.1? You may not use the `sqrt` function, and must call your function `foobar(n)`. Note that n need not be an integer, but must be positive.

Running a program on paper 6

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(0)`? _____
What is `foobar(1)`? _____
What is `foobar(-1)`? _____
What is `foobar(2)`? _____
What is `foobar(4)`? _____
What is `foobar(10)`? _____
What is `foobar(___)` for a `n` of your choice? _____

Programming on paper 7 *hard!*

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: For a given integer n, what is the smallest positive integer which is evenly divisible by all positive integers less than or equal to n? Call your function `foobar(n)`.
e.g. `foobar(4)` is 12, since 12 is divisible by 4, 3 and 2.

Running a program on paper 7

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(1)`? _____
What is `foobar(3)`? _____
What is `foobar(4.5)`? _____
What is `foobar(7)`? _____
What is `foobar(___)` for a `n` of your choice? _____

Programming on paper 8 *hard!*

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: Solve the equation $\sin(x) + \tan(x) = 1/n$ for x, to a precision of ± 0.1. Call your function `foobar(n)`.

Running a program on paper 8

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(1)`? _____
What is `foobar(3)`? _____
What is `foobar(4.5)`? _____
What is `foobar(7)`? _____
What is `foobar(___)` for a `n` of your choice? _____

Programming on paper 9

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: What is the greatest common factor of x and y? Call your function `foobar(x,y)`.

Running a program on paper 9

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(1,2)`? _____
What is `foobar(2,4)`? _____
What is `foobar(6,9)`? _____
What is `foobar(5,7)`? _____
What is `foobar(x,y)` for x and y of your choice? _____

Programming on paper 10 *very hard!*

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: Write a function to compute the probability that a determined blackjack player starting with cards totalling to p value can beat the value d without busting. Call your function `foobar(p,d)`. For simplicity, assume that each card is dealt from a fresh deck so the probability of each rank is always 1/13.

I won't explain the rules of blackjack here. You can ask me or another student if you're curious. This is not an easy problem, but it can be solved in under 40 lines of python.

A few examples:

```
foobar(20,20) = 0.0769230769231
foobar(10,19) = 0.509101852441
foobar(10,18) = 0.618446810954
foobar(6,16) = 0.576849507915
foobar(16,16) = 0.384615384615
foobar(19,18) = 1
```

Running a program on paper 10

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(20,20)`? _____
What is `foobar(19,18)`? _____
What is `foobar(___,___)` for inputs of your choice? _____

Programming on paper 11 *hard*

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: The Gregorian calendar was introduced by Pope Gregory XII in 1582. How many leap years have there been since they were introduced (the first one was in 1584), but before the year y? A leap year is a year that is evenly divisible by four, but is not also evenly divisible by 100, unless it is also evenly divisible by 400. You may call your function `foobar(y)`.

Running a program on paper 11

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(1582)`? _____
What is `foobar(1588)`? _____
What is `foobar(1590)`? _____
What is `foobar(0)`? _____
What is `foobar(2009)`? _____
What is `foobar(___)` where the input is a year of your choice? _____

Programming on paper 12

On the back side of this page, you (the programmer*) will write instructions* in python *to be followed by another student (the* computer*). That student (the computer) will follow your instructions on a given set of numbers.*

Programmer: Write a function `foobar` that given an input r returns the side of the largest square with sides having an integer length that will fit in a circle of radius r.

Running a program on paper 12

You must evaluate the following function `foobar` *manually (i.e. without using a computer). If the function will fail to execute, explain this, and suggest what python will announce. If you can fix any apparent bugs, do so and then execute the resulting code.*

What is `foobar(1)`? _____
What is `foobar(1.5)`? _____
What is `foobar(3)`? _____
What is `foobar(0)`? _____
What is `foobar(___)` where the input is a positive number of your choice?

_____

Index