A gentle introduction to COMPUTATIONAL PHYSICS with Python

Robert A. Cohen
Physics Department
East Stroudsburg University

May 25, 2026
Version 2.1

©2026 by Robert A. Cohen
Licensed under CC BY-BC-ND 4.0
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

Contents

A Computer Error **1**
A.1 The basics of kinematics 1
A.2 Loops 2
A.3 Computer resolution 4
A.4 First derivatives 6
A.4.1 First-order approach 8
A.4.2 Second-order approach 10
A.4.3 Fourth-order approach 11
A.5 Second derivatives 13
A.5.1 Third-order approach 13
A.5.2 Fifth-order approach 14

B Simultaneous Linear Equations **19**
B.1 Introduction to linear equations 19
B.1.1 A single equation with one unknown 19
B.1.2 Two equations with two unknowns 20
B.1.3 Solving by substitution 20
B.1.4 Solving by subtraction (Gaussian elimination) 22
B.2 Matrix notation 23
B.3 Gaussian elimination with the computer 25
B.3.1 A simple program 25
B.3.2 Cleaning up the code 28
B.4 Extending to n variables 30
B.4.1 Determining the value of n 30
B.4.2 Step S1: subtracting row k from the rows below to get rid of element k 31

B.4.3 Step S2: subtracting row k from all the rows above it to get rid of element k 32
B.4.4 Solving for each variable 32
B.4.5 Checking for missing first element of first row 33
B.4.6 Running the code with $n = 4$ 33
B.4.7 Limitations 34
B.5 The greenhouse effect 36
B.5.1 Radiation balance at Earth's surface 37
B.5.2 Radiation balance for the atmosphere 37
B.5.3 Greenhouse temperature in single-layer model 38
B.5.4 Extending to n atmospheric layers 38
B.5.5 Running the modified code 39
B.5.6 Analyzing the modified code 40

C Non-Linear Equations 45
C.1 Introduction 45
C.1.1 Deriving the expression 46
C.1.2 Setting up the equation 47
C.1.3 Exploring the properties of the function 48
C.1.4 The brute force method 49
C.2 Improving the method 50
C.2.1 Modified brute force method 51
C.2.2 Bisect method 52
C.3 Comparing the modified brute force and bisect method 53
C.4 Secant method 56
C.5 Newton's [aka Newton-Raphson] method 60

D Fourier Analysis 65
D.1 Reading data files and making graphs 65
D.1.1 The context 65
D.1.2 Opening the data file 66
D.1.3 Reading the data from the file 66
D.1.4 Breaking up the data into an array 67
D.2 Examining the data series for embedded functions 68
D.2.1 Determining if a particular function is present 69
D.2.2 Functions with periods that do not go evenly into the array size 71
D.2.3 Cycling through all possible sinusoidal functions 72

D.2.4 Setting the frequency domain to $n/2$ 73
D.3 Introduction to Fourier Analysis 74
D.3.1 A problem with the phase 74
D.3.2 Accounting for the phase 74
D.3.3 Determining the amplitude 75
D.3.4 Analyzing data that is "hidden" 75

E Integration **79**
E.1 Context and simple integration scheme 79
E.1.1 Numerical vs. analytical integration 79
E.1.2 Free fall 80
E.1.3 Rocket lift-off with constant thrust 81
E.1.4 Rocket lift-off with constant thrust and decreasing fuel mass 82
E.1.5 Impact of step size 85
E.2 Numerical methods of integration 86
E.2.1 Initial value method 86
E.2.2 Midrange (trapezoidal) method 87
E.2.3 Midpoint method 89
E.2.4 Taylor series expansions 91
E.2.5 Integration with higher-order Taylor series 94

F Differential Equations **99**
F.1 Introduction to differential equations 99
F.1.1 Definition of a differential equation 99
F.1.2 Analytical solutions to differential equations 100
F.1.3 Simple harmonic motion (SHM) 101
F.1.4 Using Python to plot the analytical solution 103
F.1.5 Semi-implicit Euler or Euler-Cromer scheme 103
F.1.6 Forward and Backward Euler schemes 105
F.1.7 Runge-Kutta or Heun scheme 107
F.1.8 Leapfrog scheme 107
F.2 Partial differential equations 109
F.2.1 The water wave model 109
F.2.2 The physics 109
F.2.3 The code 112
F.2.4 Simplifying the code 115
F.2.5 Improving the first derivative 116

F.2.6 Modifying the integration scheme 116
F.2.7 Modifying the integration scheme, part 2 117
F.2.8 Viscosity 118
F.3 Using VPython with Multiple Springs 119
F.3.1 Introduction to VPython 119
F.3.2 Allowing for a ball off-axis 122
F.3.3 Multiple objects 122
F.4 Derivation of Water Wave Equations 124
F.4.1 Basic Assumptions 124
F.4.2 Time Derivative of h $(\partial h/\partial t)$ 125
F.4.3 Time Derivatives of u $(\partial u/\partial t)$ and w $(\partial w/\partial t)$ 126
F.4.4 Simplifying the Equations for 1-D 127

Preface

These are readings to accompany the Computational Physics class at East Stroudsburg University. There are a couple of things that are important to keep in mind.

This class uses Python. While you can use any environment you want for creating and running Python code, these readings refer you to code written on the glowscript platform. This makes it easy to share code, and you are encouraged to write your own code, when requested, on that platform.

While it is assumed you know how to code, you do not necessarily need to be fluent in Python, as the necessary ingredients of Python will be introduced as needed. Certainly, the more Python you know, the easier it will be, but only basic Python is used so as to not present an obstacle to learning the essential elements of computational physics.

For the most part, you will not be asked to write your own code. Instead, the focus is on *interpreting* code, meaning that you need to understand why the code is structured the way it is, and on analyzing the results of the code, meaning that you need to understand why the code is producing the results so that you can debug or improve the code.

One reason for this is because, nowadays, AI can be used to write code, so not only is it difficult to assess students on their ability to write their own code but practicing physicists are less and less likely to be asked to write their own code. Instead, they need to be able to interpret and modify code that has already been written (either by AI or others).

Another reason is that this is a physics class, not a programming class. That means that the focus is on using computational methods to solve physics problems, in much the same way that your other physics classes use mathematical and graphical methods to solve physics problems. In an ideal world,

perhaps, we wouldn't need a separate computational physics class, and all physics classes would use both mathematical and computational methods to solve and gain insight into physics problems.

The readings are organized into six chapters, each chapter addressing a different class of problems. For each part, a particular physical phenomenon has been chosen that provides the context for exploring the computational methods appropriate for that class of problems. The phenomena have been selected to be ones that students are likely to be familiar (meaning they come from introductory physics) but it is expected that students will be apply the same computational techniques to all phenomena that share the characteristics of that class of problems.

These readings were developed to provide a free resource that focuses on the small subset of skills that are deemed essential for doing computational physics, thus avoiding overloading students with more than they need to know. For those interested in learning more about using Python in physics, there are several pubished books that go into more detail, and you are encouraged to explore those. Some options include texts by Wang and Landau et al, and Scopatz and Huff. One promising book, not yet published but seems to match the content in these readings, is the one by Gingrich. The book by Newman has several chapters you can download for free, along with exercises for each chapter.

A. Computer Error

- **Physics context**: Kinematics
- **Programming skills**: Python, glowscript, program comments, print options, for loops, range function, user-generated functions
- **Computation skills**: Number storage, impact of series truncation
- **Mathematics skills**: Derivatives, Taylor series expansion

A.1 The basics of kinematics

Kinematics describes the ways we can describe motion. In physics, we tend to focus on acceleration as a description of an object's motion because an object's acceleration $\vec{a}$ is proportional to the net force exerted on it $\vec{F}_{\text{net}}$, as described by Newton's second law:

$$\vec{F}_{\text{net}} = m\vec{a}$$

Acceleration is defined as the rate the velocity $\vec{v}$ changes:

$$\vec{a} = \frac{d\vec{v}}{dt}$$

And velocity is defined as the rate the position $\vec{s}$ changes:

$$\vec{v} = \frac{d\vec{s}}{dt}$$

To illustrate the relationship between acceleration and velocity, let's suppose we have an object that accelerates at 9.8 m/s^2 downward (typical free fall acceleration near Earth's surface). According to the definition of acceleration, the velocity should change by 9.8 m/s downward every second. If we multiply 9.8 m/s^2 by 20 s, we get a total velocity change of 196 m/s (downward) in 20 s.

Since this is a computational physics course, we'll start with a computer program (written in Python) that does this calculation: AFreeFall.

Try out the program. You should find it correctly determines the 196 m/s value, and that it does it two ways: one with a time step equal to 20 s and another with a time step equal to 1 s.

To see the difference, examine the code by clicking “View this program”. There are a couple of things to note.

- The code starts with `Web VPython 3.2`, which is specific to the glowscript environment. VPython is just like regular Python except that it allows for some visual elements.
- Lines that start with `#` are interpreted as comments and are ignored by the program.
- Just like in math, where you can assign values to variable abbreviations, you can do the same in Python. In this case, `9.8` has been assigned to the variable named `g`. When writing code, you can name variables whatever you want. I'm using `g` because that is the convention in physics but Python doesn't care what you use (for the most part). In fact, you don't even need to use a single letter. For example, I'm using `dt` for the time step.
- There are several lines with `v = v + g*dt`. Mathematically, such a line seems like an impossible statement since there is no value for `v` that allows both sides to be equal. However, this line is an **instruction**, not a mathematical equation. The equal sign instructs the computer to *replace* the value of `v` with the value on the right side (`v` ← `v + g*dt`) and the value of `v` is updated as a result.

A.1: What character does Python use for multiplication?

A.2: The line immediately after `dt = 1.` is `v = g*dt`. What would the code do differently if that line was `v = v + g*dt` instead of `v = g*dt`? Why?

A.3: Why are there nineteen lines with `v = v + g*dt`? What would the result be with only eighteen lines of `v = v + g*dt`? Why?

A.2 Loops

Given that the single line (with a 20-s time step) gives the same answer as the twenty lines (with a 1-s time step), it is reasonable to ask why I included the latter process in the program. There are two reasons. First, it

introduces some crucial aspects of computer programming (like that each line is an instruction, and the equal sign means "replacement" not "equivalence"). Second, it allows me to introduce some aspects of computational physics that will be utilized later, even though the value of those aspects is not obvious now.

For example, one advantage of computer programs is that they make it easy to carry out repetitive calculations. While it is not hard for me to create code that repeats the same calculation nineteen times (I just need to copy the line and then paste it eighteen times), it would be very tedious to do that for a smaller time step (for a 0.000001-s time step, I'd need to paste the line 20 million times).

Fortunately, Python provides a very easy way to do that via something called a **loop**. In this case, we want to loop through a single calculation twenty times. To do this, we can replace the twenty individual lines with the following three.

```
v = 0
for i in range(20):
  v = v + g*dt
```

A couple of things to note about these lines:

- The `v = v + g*dt` line starts with a tab character. That is very important, as I point out in another bullet.
- The `range` is called a function. It basically takes the value in parentheses (`20` in this case) and creates a series of numbers (in this case, the series starts at zero and ends at 19).
- The `for` command tells the program to repeat the lines that follow the `for` command and start with a tab. The colon `:` is important because it tells the program where the `for` command ends. The variable `i` (it needn't be an `i`) will be incremented by 1 each of the 20 times through the loop (in case you wanted to use that value within the loop).

A.4: Why is the `v = 0` line needed?

A.5: Copy the code to your own Python program and then make the replacement described above where the twenty lines are replaced by a loop (you can do this within the glowscript environment by creating an account there). Compare the calculated value of `v` that results from the loop with what you got before. Is it different? Should it be? Why or why not?

What we'd like is a way to make the program work for any time step, not just a 1-s time step. Right now, it runs the loop 20 times, but that is only appropriate for a 1-s time step. To make it more general, we can use the time step (`dt`) to determine how many times we need to go through the loop:

```
nsteps = round(20./dt,0)
```

Basically, this statement figures out how many times `dt` goes into 20 and uses the `round` function to round the answer to an integer (in case `dt` doesn't go evenly into 20). You will also need to replace `range(20)` with `range(nsteps)`, so that it works for any `dt`.

A.6: Modify your code so that it calculates the number of steps for any value of `dt` then test your code by setting `dt` to 0.01 s instead of 1 s (you can use `1e-2` instead of `0.01` if you'd like). Compare the calculated value of `v` that results from the loop with what you got before. Is it different? Should it be? Why or why not?

A.7: Modify the code so that the time step is 10^{-6}. Compare the calculated value of `v` that results with what you got before. Is it different? Should it be? Why or why not?

A.8: You should find that the program takes significantly longer to make the calculations with the two additional loops added. Why?

A.3 Computer resolution

You should find that the answer is the same regardless of the time step because the total time is 20 s every time. However, it turns out the answer is not quite the same each time. To see what I'm talking about, increase the number of digits printed out by adding the following statement to the beginning of the code.

```
print_options(digits=15)
```

Run the code again, with `dt` equal to 1 then 10^{-2} then 10^{-4} then 10^{-6} (if you want, you can put all four loops into the same code; just remember to reset the velocity to zero each time and print out the result after each loop finishes). This time, you should find that the numbers are not the same. You only get exactly 196 m/s when `dt` is equal to 1.

A.9: Which of the trials (1, 10^{-2}, 10^{-4}, or 10^{-6}) is furthest off from exactly 196 m/s?

There are two things going on here.

The first has to do with the fact that computers can't store an infinite number of digits. It is just like how we can't write out a value exactly equal to one-third in decimal. One-third is equal to $0.333\overline{3}$ (where the line over the 3 means that it is repeated). At some point we need to cut off what we write, leading to a rounding error. Computers do the same thing.

That problem, by itself, wouldn't necessarily be an issue, since it doesn't seem that any of the calculations require many digits. However, that is because we are looking at the problem in base 10. Computers store numbers in base 2, not base 10, because computers represent numbers via bits, with each bit being either 0 or 1. Numbers that are exact in decimal form may not be exact in binary. For example, one-tenth can be represented as 0.1 in decimal, but it is $0.00011\overline{0011}$ (repeating) in binary. Just like we need to cut off the digits at some point when writing one-third in decimal, the computer must cut off some digits at some point when storing one-tenth as a number in binary.

Note: For readers unfamiliar with binary, a short description of binary is provided at the end of this chapter.

Because of the rounding, the integration is a little off (for time steps less than 1 second each). Each calculation involving `g*dt` is off by a very tiny amount, but it adds up when adding a whole bunch of them together.[i]

It turns out that the computer keeps roughly 15 digits[ii]. Anything after that is rounded. You can see this yourself by changing the `print_options` digits to 16. When you try to run the code, you'll get an error. This is because the computer doesn't keep 16 digits.

A.10: Run the code with `print_options` digits set to 16. What happens?

To more clearly see what is happening, run the program AComputerTest.

[i] For the same reason, if you wanted to know the total time, it would be better to use `n*dt` rather than add up the `dt`'s, one at a time. Each `dt` is likely rounded whereas an integer like `n` is not.

[ii] The reason it only keeps 15 digits has to do with the number of bits the computer uses to represent a number (for more detail on this, see the information on binary at the end of this chapter).

This is a very simple program. Basically, it adds 10^{-16} to 1 and then subtracts 1 from the total.

You should find that this results in zero. The reason is that when 10^{-16} is added to 1 the result must be rounded to 15 digits (because of the limited space in the computer memory for a single number). That means the result is still 1. Subtracting 1 leaves you with zero.

In comparison, modify the code by adding 10^{-15} instead of 10^{-16}. You should find that this results in a non-zero number. It isn't zero because the computer was able to keep the value to the fifteenth digit.[iii]

A.11: Run the code with `testit` set to `1. + 1e-15`. What is the value of `testit` after subtracting 1? Why is it not exactly `1e-15`?

Just as there is a limited number of digits the computer can store, there is also a limit to the exponent that can be stored. As a result, the computer is unable to handle a number that is bigger than about $2 \times 10^{+308}$ or smaller than about 5×10^{-324} (see the information on binary at the end of this chapter for where these numbers come from). Anything bigger will be represented as `Infinity` and anything smaller will be equal to zero.[iv]

For example, the program AComputerTest2 shows what happens when a number is too small or too large.[v]

A.12: What do you get when you run the AComputerTest2 code and why?

A.4 First derivatives

Now that we see what is involved in getting velocity from acceleration, let's consider the reverse situation: suppose we know the velocity and want to get the acceleration?

[iii] Remember that the computer doesn't store numbers in decimal, so it isn't really 15 digits that it is storing but rather a number of digits that corresponds, in decimal, to 15 digits.

[iv] Python will also indicate a value is `Infinity` when it attempts to divide a non-zero number by zero. It indicates the value as `NaN` (not a number) when it attempts to divide zero by zero.

[v] `print_options` is set to 1 as only one digit is needed to show this.

Since the acceleration is defined as the rate at which the velocity changes, we simply need to use rules of calculus to take the time derivative of the velocity to get the acceleration.

However, how can we get the computer do it when the computer doesn't know calculus or when it is not possible to use calculus to get the derivative?

Assuming the computer knows the value of the velocity at any given time, one way it could calculate the derivative is by dividing the change in velocity by the time it takes to undergo that change. However, that only gives the correct value in certain cases. To understand when it does and doesn't work, it helps to consider the Taylor series expansion.

Any function $f(x)$ can be expanded as a Taylor series expansion as follows (where f' is the first derivative of f, f'' is the second derivative of f, and so on):

$$f(x+\delta x) = f(x) + (\delta x) f'(x) + \frac{(\delta x)^2}{2} f''(x) + \frac{(\delta x)^3}{3!} f'''(x) + \frac{(\delta x)^4}{4!} f^{(4)}(x) + \cdots \tag{A.1}$$

The terms go on forever. How many terms you include is a matter of how accurate you want to be.

By convention, a **first-order** approximation uses the terms up to the first derivative (first two terms), a **second-order** approximation uses the terms up to the second derivative (first three terms), and so on.

Important: It is very important to note that if you use the entire series then the right side will be exactly equal to the left side. In other words, you can use the terms on the right with values at x and the sum will give the value of the function at $x + \delta x$ (left side) with no error. If you only use some of the terms on the right side then you'll only get a value that is approximately the value of the function at $x + \delta x$. Some terms are larger than others, so how close you get will depend on which terms you drop. The higher-order derivative terms are likely smaller because those terms multiply by higher powers of δx (which is typically less than one) and divide by the factorial of the term number, so dropping higher-order terms likely has a smaller impact than dropping the lower-order terms.

A.13: For free fall, the first derivative is constant (9.8 m/s^2). What are the values of the second derivative, third derivative and fourth derivative?

A.14: The initial free fall program in this part had the line `v = v + g*dt`, which corresponds to only the first two terms of the Taylor series expansion (equation A.1, with `v`, `g`, and `dt` instead of f, f', and x). Did that cause our results to be off? Why or why not?

A.4.1 First-order approach

To illustrate the errors that can occur when we don't use the entire Taylor series, let's examine what happens when we try to use a truncated Taylor series to determine the derivative of sine. We'll start with the first-order approximation (only first two terms):

$$f(x + \delta x) \approx f(x) + (\delta x)f'(x)$$

We can rearrange this for $f'(x)$ as follows:

$$f'(x) \approx \frac{f(x + \delta x) - f(x)}{\delta x}$$

Since we've neglected all the terms of order 2 and higher, the error is proportional to $(\delta x)^2$. Basically, this expression is exact only if the relationship is linear (which is why we were able to use it for free fall). Otherwise, we are introducing a bias by only using the "slope" on one side of x.

This is sometimes called the **forward 2-point approach** (or **divided difference approach**). There is also the **backward 2-point approach**, which is also first order but uses the slope on the "other" side of x:

$$f'(x) \approx \frac{f(x) - f(x - \delta x)}{\delta x}$$

To illustrate, suppose the function is $\sin\theta$ and we want to determine the first derivative of this (with respect to θ) when $\theta = 1$ radians. In this case, the function depends on θ instead of x but otherwise it is the same. The program AFirstDerivative calculates the first derivative of the $\sin(1)$ using the forward first-order approach and compares that to the actual value, $\cos(1)$, given that we know that the derivative of $\sin\theta$ is $\cos\theta$.

A couple of notes about this program:

- It defines a function named `f(x)` that returns the value of `sin(x)`, which is a function in Python that calculates the sine. While it isn't necessary at this point to create our own function, it will be useful later when you need to apply the method to other functions, not just sine.
- The code sets `x` as 1 and `dx` as 0.1. This corresponds to having $\theta = 1$ radian and $\delta\theta = 0.1$ radian.
- The code includes a line that calculates the first derivative using the forward first-order approach and sets that to the variable `dfdx`.

$$\frac{d\sin(1)}{d\theta} \approx \frac{\sin(1.1) - \sin(1.0)}{0.1} = 0.497364$$

- The code prints the value of `dfdx`, the actual value (based on the cosine), and the error. In this case, the actual value is $\cos(1) = 0.540302$, meaning the error is 0.042939 (negative).

Modify the code to use $\delta\theta = 0.01$ radians. You should find that:

$$\frac{d\sin(1)}{d\theta} \approx \frac{\sin(1.01) - \sin(1.0)}{0.01} = 0.536086$$

This is off by 0.004216, which is one-tenth the error obtained when $\delta\theta$ was 0.1 radians.

Notice how using an increment one-tenth of what we used before resulted in an error that is one tenth what it was before. This is not a coincidence.

The actual error depends on the values (and signs) of the terms we've ignored but since successive terms tend to be smaller (see discussion above), most of the error is associated with the first term we ignore. In this case, that would be the second order term, $(\delta x)^2 f''(x)/2$. Since we typically don't know the value of the derivatives (for if we did we wouldn't need to do any of this), let's assume it is roughly equal to one. That means the error is $(\delta x)^2/2$, and since we divide by δx when calculating the first derivative, that means our error is roughly $(\delta x)/2$.

In our case, we first used $\delta x = 0.1$ radian, which means an error on the order of $(\delta x)/2 = 0.05$, not too far off of our actual error of 0.042939. We then used $\delta x = 0.01$ radian, which means an error on the order of $(\delta x)/2 = 0.005$, not too far off of our actual error of 0.004216.

Note: I say the error is "on the order of" because we don't know the actual error or even whether our estimate is too high or too low. Indeed, if we did,

we could fix it. All we can do is make a rough guess of how big the error might be.

A.15: Modify the code to use the backward first-order approach rather than the forward first-order approach. What is the error when using the backward first-order approach to determine $d\sin(1)/d\theta$? How does that compare to the error estimate of $(\delta x)/2$?

A.4.2 Second-order approach

Let's suppose we are unhappy with having an error on the order of δx and want one with a smaller error, like on the order of δx^2. To do that, we again use the Taylor series expansion but include all terms up to the second derivative (first three terms).

$$f(x+\delta x) \approx f(x) + (\delta x) f'(x) + \frac{(\delta x)^2}{2} f''(x) \tag{A.2}$$

... and then rearrange for $f'(x)$. We have a problem, though. This expression contains $f''(x)$ and we don't know that that is.

To fix this, we need to get rid of $f''(x)$ but without ignoring it. This is done by repeating equation (A.2) but with $-\delta x$ instead of δx.

$$f(x-\delta x) \approx f(x) - (\delta x) f'(x) + \frac{(\delta x)^2}{2} f''(x)$$

Since we are using $-\delta x$, the second term is now negative while the third term remains positive (since the square of a negative is a positive). We now have two expressions and we can subtract the second from the first to get rid of the $f''(x)$ term (we also end up getting rid of the f(x) term but that is okay):

$$f(x+\delta x) - f(x-\delta x) \approx 2(\delta x) f'(x)$$

Solving for $f'(x)$ gives:

$$f'(x) \approx \frac{f(x+\delta x) - f(x-\delta x)}{2\delta x}$$

This is sometimes called the **symmetric 3-point approach** (or the **symmetric difference approach**).

Modify the code to use the second-order with $\delta\theta = 0.1$ radians. You should find that:

$$\frac{d\sin(1)}{d\theta} \approx \frac{\sin(1.1) - \sin(0.9)}{0.2} = 0.539402$$

The actual value of the derivative is $\cos(1) = 0.0540302$. This means our error is 0.0009.

Notice how it is significantly smaller than what we had with the first order method (0.042939). Whereas the first-order method has an error on the order of $(\delta x)/2$, since the largest ignored term was $(\delta x)^2 f''(x)/2$, the second-order method has an error on the order of $(\delta x)^2/12$, since the largest ignored term is $(\delta x)^3 f'''(x)/3!$ and we are dividing by δx.[vi] Since we used $\delta x = 0.1$ radians, that means our error will be roughly $(0.1)^2/6 = 0.0017$, about twice as big as the actual error but still roughly the same order.

In general, because each successive term includes an additional power of δx, the error will be roughly $(\delta x)^n/(n+1)!$ for the nth order approximation because the first term not included is multiplied by $(\delta x)^{n+1}/(n+1)!$.

A.16: Modify the code to use second-order approach with $\delta\theta = 0.01$ instead of 0.1. How does that compare to the estimate of δx^3?

A.4.3 Fourth-order approach

Since we get better results when we use more terms, it seems reasonable to ask why we don't use all the terms. The short answer is that would be an infinite number of terms. However, even using a few more terms can become unwieldy. To illustrate, let's consider the fourth order scheme, which uses the first five terms of the Taylor series expansion.

$$f(x+\delta x) \approx f(x) + (\delta x) f'(x) + \frac{(\delta x)^2}{2} f''(x) + \frac{(\delta x)^3}{3!} f'''(x) + \frac{(\delta x)^4}{4!} f^{(4)}(x) \quad \text{(A.3)}$$

As with the second-order scheme, we need to solve this for $f'(x)$ but not have any of the other derivatives present. We can't ignore them but we can get rid of them by a process similar to what we used for the second-order

[vi] One might argue that we should also divide by 2 since the denominator in the expression is $2\delta x$, not δx, but that assumes the third-order term isn't itself multiplied by two during the derivation.

scheme. However, since there are more terms to get rid of, the process uses more equations.

Basically, we rewrite equation (A.3) for $f(x-\delta x)$:

$$f(x-\delta x) \approx f(x) - (\delta x)f'(x) + \frac{(\delta x)^2}{2}f''(x) - \frac{(\delta x)^3}{3!}f'''(x) + \frac{(\delta x)^4}{4!}f^{(4)}(x)$$

Then we subtract this from equation (A.3) to get rid of the f, f'' and $f^{(4)}$ terms:

$$f(x+\delta x) - f(x-\delta x) \approx 2(\delta x)f'(x) + \frac{2(\delta x)^3)}{3!}f'''(x) \tag{A.4}$$

Then we rewrite this expression for $f(x+2\delta x)-f(x-2\delta x)$, which is equivalent to just rewriting it and replacing δx by $2\delta x$:

$$f(x+2\delta x) - f(x-2\delta x) \approx 4(\delta x)f'(x) + \frac{16(\delta x)^3}{3!}f'''(x) \tag{A.5}$$

We can now get rid of the f''' term by multiplying the equation (A.4) by 8, subtracting equation (A.5) from that and then solving for $f'(x)$ to get the fourth-order scheme:

$$f'(x) \approx \frac{f(x-2\delta x) - 8f(x-\delta x) + 8f(x+\delta x) - f(x+2\delta x)}{12\delta x}$$

This is sometimes called the **5-point approach**.

A.17: Show (write out the algebra) how one gets from equations (A.4) and (A.5) to the fourth-order scheme.

To show how well this works, modify the code to use the fourth-order approach with $\theta = 0.1$ radian. You should get the following:

$$\frac{d\sin(1)}{d\theta} \approx \frac{\sin(0.8) - 8\sin(0.9) + 8\sin(1.1) - \sin(1.2)}{1.2} = 0.540301$$

The error is around 1.8×10^{-6} (roughly the same as the rounding error). Notice how it is significantly smaller than what we had with the first-order method (0.042939) and second-order method (0.0009) and is consistent with the fourth order method being off by around $\delta x^4/5!$ (8.3×10^{-7}).

Using more terms is more accurate but it requires more work to get the appropriate equation and at some point the added accuracy isn't worth the additional effort.

A.18: Modify the code so that the function is $x^4 + 3x^3 - 5x^2 + x$ instead of $\sin\theta$, and compare the actual first derivative at $x = 10$ with what you get when you use (a) the first-order forward approximation with $\delta x = 1$, (b) the second-order approximation with $\delta x = 1$, and (c) the fourth-order approximation with $\delta x = 1$. To raise something to a power, use two asterisks. For example, `3*(x**3)` would be equivalent to $3x^3$.

A.19: Suppose it was possible to derive a sixth-order approximation. For the function in the previous problem, predict the error that would result if you used the sixth-order approximation and explain your reasoning.

A.5 Second derivatives

Now that we see how the computer can determine the first derivative, it is reasonable to ask how the computer can determine the second derivative (equivalent to determining the acceleration from the position). The answer is to again use the Taylor series approximation multiple times and combine in such a way that we get rid of every term except the second derivative. Here I will show you two approaches.

Note: Since we want an expression for the second derivative, we need at least the second order approximation.

A.5.1 Third-order approach

The third-order approach uses the Taylor series out to third order (neglecting all terms of order 4 and higher, so that the error is proportional to δx to the fourth power):

$$f(x + \delta x) \approx f(x) + (\delta x) f'(x) + \frac{(\delta x)^2}{2} f''(x) + \frac{(\delta x)^3}{3!} f'''(x)$$

Since we've neglected all the terms of order 4 and higher, the error is proportional to δx^4. To get the second derivative, we repeat the expression for $x - \delta x$ as follows:

$$f(x - \delta x) \approx f(x) - (\delta x) f'(x) + \frac{(\delta x)^2}{2} f''(x) - \frac{(\delta x)^3}{3!} f'''(x)$$

We then add the two together to get

$$f(x+\delta x)+f(x-\delta x)\approx 2f(x)+(\delta x)^2 f''(x)$$

Solve to get:

$$f''(x)\approx \frac{f(x-\delta x)-2f(x)+f(x+\delta x)}{(\delta x)?^2}$$

This is the **three-point method** and the one most people use.

A.5.2 Fifth-order approach

The fifth-order approach uses the Taylor series out to fifth order (neglecting all terms of order 6 and higher, so that the error is proportional to δx to the sixth power):

$$f(x+\delta x)\approx f(x)+(\delta x)f'(x)+\frac{(\delta x)^2}{2}f''(x)+\frac{(\delta x)^3}{3!}f'''(x)+\frac{(\delta x)^4}{4!}f^{(4)}(x)+\frac{(\delta x)^5}{5!}f^{(5)}(x)$$

We want an expression for $f''(x)$. As before, rewrite the expression for $f(x-\delta x)$:

$$f(x-\delta x)\approx f(x)-(\delta x)f'(x)+\frac{(\delta x)^2}{2}f''(x)-\frac{(\delta x)^3}{3!}f'''(x)+\frac{(\delta x)^4}{4!}f^{(4)}(x)-\frac{(\delta x)^5}{5!}f^{(5)}(x)$$

We then add the second from the first:

$$f(x-\delta x)+f(x+\delta x)\approx 2f(x)+(\delta x)^2 f''(x)+\frac{(\delta x)^4}{12}f^{(4)}(x)$$

Then we rewrite this expression for $f(x+2\delta x)-f(x-2\delta x)$, which is equivalent to just rewriting it and replacing δx by $2\delta x$:

$$f(x-2\delta x)+f(x+2\delta x)\approx 2f(x)+4(\delta x)^2 f''(x)+\frac{4(\delta x)^4}{3}f^{(4)}(x)$$

We can now get rid of the $f^{(4)}$ term by multiplying the first version by 16 and then subtracting the second from the first:

$$16f(x-\delta x)+16f(x+\delta x)-f(x-2\delta x)-f(x+2\delta x)\approx 30f(x)+12(\delta x)^2 f''(x)$$

Solving for $f''(x)$ we get:

$$f''(x) \approx \frac{-f(x-2\delta x) + 16f(x-\delta x) - 30f(x) + 16f(x+\delta x) - f(x+2\delta x)}{12(\delta x)^2}$$

This is the **five-point method**.

A.20: Suppose you modified your code to calculate the third-order and fifth-order approximations to the second derivative of $x^4 + 3x^3 - 5x^2 + x$ at $x = 10$ with $\delta x = 0.1$, and compared them with the exact value (using calculus). Why would the fifth-order approximation be exact but the third-order approximation be off?

Binary numbers

We use the **decimal** numeral system, which uses ten different numbers from 0 to 9. Since each digit has ten possible values, each digit represents a factor of ten. So, for example, a number like 123 can be expressed as $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$.

In comparison, a **binary** numeral system uses only two different numbers, 0 and 1. Since each digit has two possible values, each digit represents a factor of two. So, for example, a number like 111 can be expressed as $1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$. Computers use binary because the electronics are based on voltages being in one of two states, like high and low.

Every integer can be expressed exactly, either in decimal or binary, but with binary many more digits are needed. For example, a binary number consisting of eleven 1's (11111111111) is equal to 2045, which only requires four digits in decimal. A binary number consisting of 52 digits in binary can be represented by only 16 digits in decimal.

Most computers use 64 binary digits to express a number[vii]. Each digit is call a **bit**. One bit is used to indicate the sign (positive or negative). The remaining are used to represent the number, and are split into two groups: eleven for the exponent and 53 for the fraction. This limits the number of integers that can be represented. In particular, the largest number is around 1.8×10^{308} (2^{1024}) and the smallest number is around 5×10^{-324} (2^{-1074}). This

[vii] See the Wikipedia entry on double-precision floating-point format for more information

is because the eleven digits for the exponent can express powers of two from 2^{-1022} to 2^{+1023} (since eleven bits can be used for 2045 values, as indicated above[viii]), and the 53 digits for the fraction can express powers of two from 2 down to 2^{-52}.

Notice that numbers with absolute value greater than 1 can be expressed exactly in binary but numbers with absolute value less than 1 can be exact in decimal but not exact in binary. For example, fractions are represented as negative powers of 2. A fraction like one-sixteenth can be represented exactly in binary but a fraction like one-tenth cannot be, and will be approximated by the number of digits available (53), which corresponds to 16 digits in decimal. Some numbers, like one-third, cannot be represented exactly in either binary or decimal.

Problems

Problem A.1: The following is some sample code for calculating the speed of an object in free fall.

```
print_options(digits=15)
v = 0.
for t in range(0,10.,2.**-4):
  v = v + 9.8*2.**-4
print ("final velocity =",v,"m/s")
```

When this code is run, it produces the output

```
final velocity = 97.9999999999997 m/s
```

Why is the final velocity not `98 m/s`? Choose one of the following and provide your rationale.

A. 9.8 is not represented exactly in binary.
B. 2^{-4} is not represented exactly in binary.
C. The range function should start at 0.01, not 0.
D. The range function needs to have a single parameter, not three.
E. `v = 9.8*2.**-4` needs to be added prior to the loop.

Problem A.2: Suppose we took our code for free fall in Problem #1 and wanted to modify it to include drag, with an amount equal to $-bv$, where $b = 0.01$ s^{-1}. Which line or lines would we have to modify, and why?

[viii] An exponent value of zero is reserved for NaN and Infinity

Problem A.3: Suppose we used our free fall program in Problem #1 with a time step of δt and wanted to predict the change in velocity over a period equal to Δt. Predict the value that would result with each of the following values of δt and Δt, and explain your reasoning for each prediction:

- $\delta t = 10^{-16}; \Delta t = 10^{-13}$
- $\delta t = 10^{-116}; \Delta t = 10^{-113}$
- $\delta t = 10^{-216}; \Delta t = 10^{-213}$
- $\delta t = 10^{-416}; \Delta t = 10^{-413}$

Problem A.4: The Taylor series expansion is given in equation A.1.

$$f(x+\delta x) = f(x) + (\delta x) f'(x) + \frac{(\delta x)^2}{2} f''(x) + \frac{(\delta x)^3}{3!} f'''(x) + \frac{(\delta x)^4}{4!} f^{(4)}(x) + \cdots$$

Suppose we used the third-order approximation to find the second derivative of the following function:

$$f(x) = x^2 e^{2x}$$

What would be the approximate error if we used $\delta x = 0.01$? Explain how you obtained your estimate.

Problem A.5: We derived an expression for the first order derivative, the second order derivative and the fourth order derivative. Suppose we wanted to calculate the derivative of a quadratic expression. Would any of those expressions give us an answer exactly equal to the analytical value? If so, which ones and why? If not, why not?

B. Simultaneous Linear Equations

- Physics context: Greenhouse effect
- Programming skills: Arrays, if statements, len, input, float, append
- Computational skills: Gaussian elimination
- Mathematical skills: Matrices

B.1 Introduction to linear equations

The average surface temperature on Venus is 737 K, which is hotter than the average surface temperature on the sunlit side of Mercury (590-725 K). This is due to the presence of a thick atmosphere on Venus that absorbs radiation from the surface that would otherwise radiate out to space. A proper modeling of this so-called greenhouse effect treats the atmosphere as many layers, with each layer being in radiative equilibrium, meaning each layer absorbs as much radiation as it emits. For n layers, we'll have n relationships with n temperatures, one for each layer. The focus in this part is to use the computer to solve for those n temperatures.

First, though, we'll examine a much simpler situation, one where we only have one equation and one unknown, much like what you've encountered in an algebra class. Then we'll move onto two equations and two unknowns, also something you've probably encountered in an algebra class. We'll walk through the process you used in algebra class and then we'll introduce some matrix notation (section B.2), which will make it easier to solve with a computer program (in section B.3) and ultimately to more than two equations (in section B.4) and then to the greenhouse effect (in section B.5).

B.1.1 A single equation with one unknown

A linear equation is an equation of one unknown (called a variable) that does not include any powers of that unknown. An example of a linear equation is

as follows:

$$2x + 3 = 8$$

To obtain the value of the unknown (x in this example), we'd subtract 3 from both sides, to get the x term by itself. This gives $2x = 5$. We would then divide both sides by 2, to get x by itself. This leaves $x = 2.5$, which is the solution.

Notice how we first add or subtract to get the x term by itself then multiple or divide to get x by itself.

B.1.2 Two equations with two unknowns

The following is an example of a set of two equations with two variables x and y:

$$\begin{aligned} 2x + 3y &= 8 \\ 5x - 2y &= 1 \end{aligned} \tag{B.1}$$

Notice that we can't solve to get the values of x and y using just one or the other. For example, using the technique discussed above, we'd get two equations for x but they'd each depend on y and we don't know the value of y.

In general, there are two approaches we can use, both of which require us to somehow combine both equations (which amounts to using both equations simultaneously). The first approach, which is called **solving by substitution**, is the one you most likely used in an algebra class. The second approach, which is called **solving by subtraction** (or **Gaussian elimination**), is better suited for our computational approach, which we'll examine in section B.3.

B.1.3 Solving by substitution

Solving by substitution involves taking one of the equations, rearranging it for one variable in terms of the other, then plugging that rearranged equation into the other equation. For example, we can rearrange the first equation in (B.1) as follows by using the same two-step process we used before for single equations:

$$x = \frac{8 - 3y}{2}$$

Then we take $(8-3y)/2$ and plug that in for x in the second equation (since the two x's must have the same value).

$$5\left(\frac{8-3y}{2}\right) - 2y = 1$$

This gives us a single expression with just one unknown: y. We then solve this equation for y by first combining the y terms then solving using the same two-step process we used before for single equations:

$$\begin{aligned} 20 - 19y/2 &= 1 \\ -19y/2 &= -19 \\ y &= 2 \end{aligned}$$

Finally, we then take that value of y and plug it into the first equation to solve for x (using the same two-step process we used for the single equation):

$$\begin{aligned} 2x + 3 \cdot 2 &= 8 \\ x &= 1 \end{aligned}$$

B.1: The illustration above solved the first equation for x and then plugged that into the second equation to solve for y. Would the result (values of x and y) be any different if we had solved the first equation for y (instead of x) and then plugged that into the second equation to solve for x? What about if we start by solving the second equation for x or y, instead of the first equation?

A couple of things to notice:

- With two variables, like x and y, we wouldn't be able to solve for either one if we only had a single equation relating them. And if we had three variables, x, y and z, we wouldn't be able to solve for them with only two equations. If we have n variables, we'd need at least n equations.
- Each of the n equations need not contain all n variables. We only need the n variables to appear somewhere in the n equations, as with the following equations (which can be solved for x and y):

$$\begin{gathered} 2x + 3y = 8 \\ y = 2 \end{gathered}$$

- Just because we have n equations doesn't mean we'd necessarily be able to solve for the n variables. The n equations need to be **independent**. An equation is independent when we can't get it from some combination of the others. For example, we wouldn't be able to solve for x and y if the second equation is just some multiple of the first, like the following:

$$\begin{aligned} 2x + 3y &= 8 \\ 4x + 6y &= 16 \end{aligned} \tag{B.2}$$

- If we have *more* than n independent equations with n variables then any n of those equations would be sufficient to solve for the n variables. For example, any two of the following equations could be solved for x and y.

$$\begin{aligned} 2x + 3y &= 8 \\ 5x - 2y &= 1 \\ 2x - 3y &= -4 \end{aligned} \tag{B.3}$$

B.2: It was mentioned that the equations in (B.2) cannot be solved for x and y because the two equations are not independent (meaning we can get the second from the first). Try to solve for x by using the substitution method described earlier. What is the result and why can't that be used to find x?

B.3: Show how the second two equations in (B.3) can be used to find x and y.

B.1.4 Solving by subtraction (Gaussian elimination)

The alternate approach is to subtract the two equations to remove one of the variables. This approach is called **Gaussian elimination**.[i]

To illustrate, again consider the two equations in (B.1) from before, which I'll renumber as (B.4) and (B.5):

$$2x + 3y = 8 \tag{B.4}$$

$$5x - 2y = 1 \tag{B.5}$$

There are two basic steps in the solution, which I label S1 and S2.

[i]This method is named after Gauss because of the matrix notation he devised to carry it out, a notation similar to the matrix notation we'll be using later. However, the method itself was not originated by Gauss and was probably common knowledge in the time of Gauss (it was written out by Newton, for example).

S1 The first step is to get an expression containing only y.

i) First, *scale* the first equation so that both equations have same x coefficient. In our example, this means multiplying equation (B.4) by a factor of 5/2 to get:

$$5x + 15y/2 = 20 \tag{B.4a}$$

ii) Then subtract this new equation (B.4a) from the second equation (B.5).

$$-19y/2 = -19 \tag{B.5a}$$

S2 The second step is to get an expression containing only x.

i) First *scale* the equation (B.5a) so that the y coefficient is the same as in equation (B.4a). In our example, this means multiplying equation (B.5a) by $-15/19$ to get:

$$15y/2 = 15 \tag{B.5b}$$

ii) Then subtract this equation (B.5b) from equation (B.4a).

$$5x = 5 \tag{B.4b}$$

The process gives two equations, each with just a single variable, and we can solve for each by dividing through by the coefficients, giving the same result as before ($x = 1$, $y = 2$).

The subtraction method may seem overly cumbersome but it has the advantage of being easier to program (as you'll see in section B3) since we don't have to switch the location of the variables from one side of the equal sign to the other.

B.4: Show how one can solve the following set of equations using Gaussian elimination.

$$x - y = -1$$
$$2x - 3y = -4$$

B.2 Matrix notation

Rather than write out the two equations in the traditional form, one below the other, it is easier to use **matrix notation**, where the coefficients are

written as a matrix. This is illustrated below, where the equations in (B.1) are expressed in matrix form:

$$\begin{bmatrix} 2 & 3 \\ 5 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ 1 \end{bmatrix}$$

Notice how the top row of the left matrix contains the coefficients of x and y for the first equation, and the top row of the right matrix indicates the value of the right side of the first equation. The second row of each matrix corresponds to the second equation.

To show how well this notation works with Gaussian elimination, let's repeat the solution from the previous section but with the new notation.

S1 The first step gets an expression containing only y. In matrix notation, this is equivalent to getting rid of the first element of the second row.

i) First *scale* the first row (corresponding to the first equation) so that both rows have same x coefficient (first column). This means multiplying the top row by a factor of $5/2$:

$$\begin{bmatrix} 5 & 15/2 \\ 5 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 20 \\ 1 \end{bmatrix}$$

ii) Then subtract the first row from the second row.

$$\begin{bmatrix} 5 & 15/2 \\ 0 & -19/2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 20 \\ -19 \end{bmatrix}$$

S2 The second step is to get an expression containing only x. In matrix notation, this is equivalent to getting rid of the second element of the first row.

i) First scale the second row so that both rows have the same y coefficient (second column). This means multiplying the bottom row by $-15/19$ to get:

$$\begin{bmatrix} 5 & 15/2 \\ 0 & 15/2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 20 \\ 15 \end{bmatrix}$$

ii) Then subtract the bottom row from the top row.

$$\begin{bmatrix} 5 & 0 \\ 0 & 15/2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \end{bmatrix}$$

We now have two equations, each with just a single variable, and we can solve for each by dividing through by the coefficients, giving the same result as before ($x = 1$, $y = 2$).[ii]

B.5: Solve the following set of equations using Gaussian elimination. Explain what you do at each step.

$$\begin{bmatrix} 1 & 4 \\ 5 & -3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -7 \\ 1 \end{bmatrix}$$

B.3 Gaussian elimination with the computer

B.3.1 A simple program

A simple linear equation solver program is F:SimpleLinearSolver.

Try out the program. It solves the two linear equations shown below (from equation B.1 in section B.1):

$$\begin{aligned} 2x + 3y &= 8 \\ 5x - 2y &= 1 \end{aligned}$$

In matrix form, these two equations are written as follows:

$$\begin{bmatrix} 2 & 3 \\ 5 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ 1 \end{bmatrix} \tag{B.6}$$

From section B.1 we know that the solution to this set of equations is $x = 1$ and $y = 2$. When you run the code, it should print out a bunch of stuff, the last line of which provides the solution of $[1, 2]$, which corresponds to $x = 1$ and $y = 2$.

Let's go through each part of the program and decipher what it is doing and why. Note that there are several print statements scattered throughout the code. These are there so you can see what is being done to the matrices at each step. I won't be discussing those print statements here.

[ii] In matrix parlance this is called **diagonalizing** the matrix.

The first part of the code assigns the matrix values:

```
A = [ [2., 3.], [5., -2.]  ]
B = [8., 1.]
```

Notice how `A` contains information from the 2×2 matrix on the left side of equation (B.6) and `B` represents the information from the right matrix.[iii]

To refer to each element of the matrix you need two indices, one for the row (each row corresponding to an equation) and one for the column (each column corresponding to a variable coefficient) as follows: `A[i][j]`. Python starts indices at zero so `A[0][0]` corresponds to the first variable coefficient of the first equation (2 in our case) and `A[0][1]` corresponds to the second variable coefficient of the first equation (3 in our case).

B.6: For the set of equations in (B.6), what is the value of `A[1][0]` and `B[1]`?

As discussed in section B.1, there are two basic steps in the solution, which I indicated as steps S1 and S2.

S1 The first step gets rid of the first element of the second row.

i) First multiply the first row by a factor such that the first element in that row is equal to the first element in the second row.

```
factor = A[1][0] / A[0][0]
B[0] = B[0] * factor
A[0][0] = A[0][0] * factor
A[0][1] = A[0][1] * factor
```

ii) Next subtract these values from the values in the second equation.

```
B[1] = B[1] - B[0]
A[1][1] = A[1][1] - A[0][1]
A[1][0] = A[1][0] - A[0][0]
```

Note that the last line is unnecessary as we already know `A[1][0]` will be zero. That is why we subtracted. Consequently, we could just set `A[1][0]` to zero (or just ignore it entirely). However, it is

[iii] In regular python this notation assigns the values as “lists” rather than “real” arrays. If you were to use regular python (not glowscript), it would be more computationally efficient to import the numpy library and use the `array` command:

```
from numpy import *
A = array ( [ [2., 3.], [5., -2.]  ]  )
```

In glowscript, there is no advantage to doing this (and numpy is available as default).

kept in the program so that you can track its value (via the print statements) and ensure that `A[1][0]` does indeed equal zero after the subtraction.

S2 The second step is to get rid of the second element of the first row.

i) First multiply the second row by a factor such that the second element in that row is equal to the second element in the first row.

```
factor = A[0][1] / A[1][1]
B[1] = B[1] * factor
A[1][0] = A[1][0] * factor
A[1][1] = A[1][1] * factor
```

Again, note that the line for `A[1][0]` is not necessary since it should be zero, but it is included so you can track its value.

ii) Next subtract this second row from the first row (and replace the first row with the results).

```
B[0] = B[0] - B[1]
A[0][0] = A[0][0] - A[1][0]
A[0][1] = A[0][1] - A[1][1]
```

In this case, the expressions for `A[0][0]` and `A[0][1]` are unnecessary because `A[0][0]` should remain unchanged (since the first element in the second expression should be zero) and `A[0][1]` should end up being zero. I include them here for completeness.

At this point we are left with two equations, each with a single variable, and we can solve for the value of each variable.

```
B[0] = B[0] / A[0][0]
B[1] = B[1] / A[1][1]
```

B.7: In step S2(i) we multiply the second row of elements by some factor prior to subtracting from the first row. Could we, instead of multiplying the second row of elements by some factor, have multiplied the first row of elements by some, perhaps different, factor (and then subtracting the second row from the revised first row)?

B.8: Copy the code to your own folder so you can edit it. Show that the code works with a different set of linear equations. Change at least one element of `A` and change the values in the `B` matrix accordingly (since changing `A` will also change `B`). Make sure the values you choose for the two matrices contain no zeroes, have a solution, and the solution is not the null set (zeroes). What `A` and `B` matrix values did you use and what solution did the code produce?

B.9: Is the solution you obtained in problem B.8 what you expected? Why or why not?

B.3.2 Cleaning up the code

The code is rather simple and can fail even when, mathematically, we know a solution exists.

For example, the code fails if a zero exists in the first index (meaning that there is no x value in the first or second equation). For example, in the original code, the solution had $y = 2$, so let's consider what would happen if the second equation was just $y = 2$. Instead of the original matrix formulation:

$$\begin{bmatrix} 2 & 3 \\ 5 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ 1 \end{bmatrix}$$

we'd have:

$$\begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ 2 \end{bmatrix} \tag{B.7}$$

Mathematically, the result is the same. However, the program would fail because it would scale the first equation by multiplying by zero (to make the first coefficients match) and then the first row would just be all zeroes, and we'd end up with just one equation (represented by the second row).

B.10: Show that the code doesn't work when `A[1][0]` is zero. Set the values of `A` and `B` to match those in equation (B.7). What `A` and `B` matrix values did you use (write down those two lines of code), what solution did the code produce, and what is it about the code that makes it fail?

To address this, we'll modify the code so that it doesn't update the first row at all during step S1. Instead, we'll have it figure out what the coefficients of the first row would be when scaling it and then subtract that from the coefficients in the second row. To do this, replace the seven lines that carry out steps S1(i) and S1(ii) with the following four lines:

```
factor = A[1][0] / A[0][0]
B[1] = B[1] - B[0] * factor
A[1][1] = A[1][1] - A[0][1] * factor
A[1][0] = A[1][0] - A[0][0] * factor
```

Note that this updates `A[1][0]` and `A[1][1]` while leaving `A[0][0]` and `A[0][1]` alone. You can delete the print statement for the "Intermediate solution after scaling first row:" since the program no longer does that step separately.

Do the same for the parts of step S2 so that it updates `A[0][0]` and `A[0][1]` while leaving `A[1][0]` and `A[1][1]` alone. Make sure the code now works and gives the same result as before ($x = 1$, $y = 2$). While this change fixes the error that results when the first coefficient of the second row is zero, the program still has a problem when the first coefficient of the *first* row is zero (i.e., the first equation doesn't include x) because the program would attempt to divide by that number (to figure out the first factor).

B.11: Show that the code doesn't work when `A[0][0]` is zero. Flip the two rows so that the first row corresponds to $y = 2$ and the second row corresponds to $2x + 3y = 8$. What `A` and `B` matrix values did you use (write down those two lines of code), what solution did the code produce, and what is it about the code that makes it fail?

We can avoid this error by flipping the rows whenever we encounter a zero in the first coefficient of the first row. Rather than implement this solution, we'll just check for it and, if we find it is zero, print out that an error exists, using the following code prior to doing step S1:

```
if A[0][0] == 0:
  print ("ERROR: element",0,"of row",0,"is zero")
```

It isn't terribly difficult to modify the program to flip the rows. However, we won't be doing that because flipping rows can make the code unnecessarily complicated, especially when we extend the program to handle more than two equations (since we'll need to check for each row). Using a check, instead of a flip, also provides a nice excuse to learn about if statements. Notice how the `if` statement checks if `A[0][0] == 0` is identical to zero. Also notice the colon `:` at the end and the indentation of the `print` statement that follows. Those are very important as they tell Python what to do when the check is true.

Keep in mind that the code will also fail if the equations are not independent. For example, the following set of equations are not independent:

$$x + y = 2$$
$$2x + 2y = 4$$

B.12: Set up your program to solve the two equations shown above. It should fail. What procedure is the program attempting that leads to the failure?

Even if the two equations are not independent, the program can still fail simply because there is no mathematical solution. For example, the following set of equations has no solution, as you should be able to readily see, since the coefficients in the second equation are just double that of the first but the right side is not double:

$$x + y = 2$$
$$2x + 2y = 3$$

B.13: Set up your program to solve the two equations shown above. It should fail. What procedure is the program attempting that leads to the failure?

B.4 Extending to n variables

While it is nice to have a program that solves a set of two equations and two variables, there is not much use for such a program because it is relatively easy to solve a set of two equations and two variables. What is more valuable is a program that solves a set of three or more equations, as it gets harder and more time consuming to solve by hand the more equations we have.

To make the code more powerful (and useful), we'll modify our program to work with any number of linear equations (which will allow us to model the atmosphere's greenhouse effect; see section B.5). Before moving on, make a copy of your code that you can then edit for this section.

B.4.1 Determining the value of n

Since the code should work for any number of equations, we need a way for the program to determine how many equations there are. For this, we'll use the `len` function. This function determines the size of an array, like `B`. For the two equations we examined in part B.3, `N` would be 2. Go ahead and add the following statement after the two lines at the beginning of your program where you specific the values of `A` and `B`.

```
N = len(B)
```

B.4.2 Step S1: subtracting row k from the rows below to get rid of element k

With a 2×2 matrix, the first step in Gaussian elimination is to multiply the first row by some factor (so that when we subtract that row from the second row we get rid of the first element in the second row). With n rows (of n elements each), we need to do the same thing but for every row (except for the last since the last row doesn't have any rows below it). That means we need to replace the four lines of step S1 with six lines. The first two lines are as follows and tell the program to go through each row.

```
for k in range(N-1):  # the row we are subtracting
  for i in range(k+1,N): # each row below k that we are
subtracting from
```

Notice that `k` represents the row that we are subtracting and `i` represents the loop that we are subtracting from. Also recall that the program ignores everything after the `#`, so everything after the `#` only serves to help us understand the code (these are referred to as **comments**).

Note also that the second `for` loop uses `range(k+1,N)`, which has two parameters, rather than just one parameter like `range(N-1)` with the first `for` loop. Recall from chapter A that the range function creates a series of numbers starting at zero and ending at one less than the number specified. That means that `range(N-1)` creates a series of numbers from zero to `N-2`. Using two parameters instead of one tells the range function where to *start* the series, as well as where to *end* it. Consequently, `range(k+1,N)` creates a series of numbers starting at `k+1` and ending at `N-1`.

Within the loop, we need to figure out the factor to multiply and then subtract the scaled values of the row `k` elements from the elements in row `i`. Remember that lines within a loop must be indented.

```
    factor = A[i][k] / A[k][k]
    B[i] = B[i] - B[k] * factor
    for j in range(k,N): # each column of row i
      A[i][j] = A[i][j] - A[k][j] * factor
```

B.14: With two equations (as in section B.3), `N` would be 2. For the first loop, how many values of `k` should it go through when `N` is 2? For the second loop, how many values of `i` should it go through when `N` is 2? For the third

loop, how many values of j should it go through when N is 2? Recall that the range function was described in chapter A.

B.4.3 Step S2: subtracting row k from all the rows above it to get rid of element k

As with step S1, we need to replace the four lines of step S2 with six lines.[iv]

```
for k in range(1,N): # the row we are subtracting
  for i in range(k):  # each row above k to subtract from
    factor = A[i][k] / A[k][k]
    B[i] = B[i] - B[k] * factor
    for j in range(k,N): # each column of row i
      A[i][j] = A[i][j] - A[k][j] * factor
```

With a 2×2 matrix, the second step in Gaussian elimination was to multiply the second row by some factor such that when we subtract that row from the first row we get rid of the second element in the first row. With n rows (of n elements each), we need to do the same thing but for every row, which is why we are using the for loops, as with step S1. The difference is that for step S2 the first loop (k) doesn't evaluate the first row since the first row doesn't have any rows above it, which is why the first range is (1,N) instead of (N-1). In addition, since we are subtracting rows above rather than below, the i loop goes from the first row (0) to the row above k, rather than from k+1 to the last row. The subtraction process (j loop) is the same as in step S1.

B.4.4 Solving for each variable

The last step is to divide the values in B with the corresponding diagonal elements in A. This is just another loop and is left as an exercise for you (see Question B.15).

[iv] The first two loops could instead use range(N-1,0,-1) and range(k-1,-1,-1). The third parameter in the range function specifies the step, and since it is -1 that means it sequences down rather than up.

B.4.5 Checking for missing first element of first row

Recall that the first thing our simple program did was check if element `A[0][0]` is zero and, if it is zero, let us know. With more than two rows we should still do a check but we need to do it for every diagonal element `A[k][k]`, not just the first. If you don't do the check, the program will still run but you'll end up with `NaN` values[v] and won't know what went wrong.

To check for each line, revise the two lines as follows:

```
    if A[k][k] == 0:
      print ("ERROR: element",k,"of row",k,"is zero")
```

You also need to move the check to within steps S1 and S2 right before the program divides by `A[k][k]`. You can't check before steps S1 and S2 because the diagonal values can change during the course of those steps (as rows are subtracted from other rows).

B.15: Run your code with the values in equation (B.1) and verify that you obtain the same results (x=1, y=2). What code did you use for the last part of the program, where you divide the values in `B` with the corresponding diagonal elements in `A`?

B.4.6 Running the code with $n = 4$

Run your revised code with the following parameters (the backslash \ indicates that the statement continues onto the next line):

```
A = [ [4., 1., 2., -3], [-3., 3., -1., 4.],\
  [-1., 2., 5., 1.], [5., 4., 3., -1] ]
B = [-16., 20., -4., -10.]
```

This corresponds to following set of equations:

$$\begin{aligned} 4w + x + 2y - 3z &= -16 \\ -3w + 3x - y + 4z &= 20 \\ -w + 2x + 5y + z &= -4 \\ 5w + 4x + 3y - z &= -10 \end{aligned} \tag{B.8}$$

[v] `NaN` stands for "not a number." `Infinity` is when python tries to divide a non-zero number by zero, whereas `NaN` is when python tries to divide zero by zero (or multiply zero by `Infinity`).

You should find that the step S1 process removes all the elements below the diagonal (makes them all zero) while the step S2 process removes all the elements above the diagonal (so the result just has non-zero values along the diagonal). This set of equations should have a solution equal to −1, 1, −2, and 3. Do not move on until you confirm that you obtain these values.

B.4.7 Limitations

We've already identified one limitation of this method in that it can't deal with values of zero along the diagonal. That doesn't mean there isn't a solution. It just means that the program, as designed, can't find it. For example, the solution to the above set of equations is $w = -1$, $x = 1$, $y = -2$, and $z = 3$. That means we can replace any of the equations with, say, $z = 3$, and we'd have the same solution. For example, if we replaced the first equation in set (B.8) with $z = 3$, the four equations would be:

$$\begin{aligned} z &= 3 \\ -3w + 3x - y + 4z &= 20 \\ -w + 2x + 5y + z &= -4 \\ 5w + 4x + 3y - z &= -10 \end{aligned} \tag{B.9}$$

Without the program, we can easily see that we could simply plug 3 in for z and we'd get three equations and three variables, as shown below, which we can solve:

$$\begin{aligned} -3w + 3x - y &= 8 \\ -w + 2x + 5y &= -7 \\ 5w + 4x + 3y &= -7 \end{aligned} \tag{B.10}$$

In fact, the program is perfectly fine solving set of three equations and three variables (set B.10). However, it bombs when doing the equivalent set of four (set B.9), even though the solution is the same.

$$\begin{bmatrix} -3 & 3 & -1 \\ -1 & 2 & 5 \\ 5 & 4 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 8 \\ -7 \\ -7 \end{bmatrix} \quad \text{vs.} \quad \begin{bmatrix} 0 & 0 & 0 & 1 \\ -3 & 3 & -1 & 4 \\ -1 & 2 & 5 & 1 \\ 5 & 4 & 3 & -1 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ 20 \\ -4 \\ -10 \end{bmatrix}$$

B.16: Run your code with the $N = 3$ set on the above left. Is the solution provided by the program what you expected based on the discussion above? Why or why not?

B.17: Run your code with the $N = 4$ set on the above right. Is the solution provided by the program what you expected based on the discussion above? Why or why not?

A related problem occurs when the program "creates" a zero along the diagonal during the subtraction process. For example, consider the following set of equations:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \\ 2 \end{bmatrix} \tag{B.11}$$

The solution to set (B.11) is $x = 1$, $y = 2$, and $z = 3$, as you can confirm simply by plugging those values into each equation and showing that they work. However, because the program is using Gaussian elimination, the moment it subtracts the first row from the second row, the result will be $0\ 0\ -2$, which has zero along the diagonal and thus the program blows up.

If the program was smart enough, it could avoid this problem simply by switching the second and third rows, as follows:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \\ 0 \end{bmatrix} \tag{B.12}$$

B.18: Try running the program with matrix values in (B.11). Is the solution provided by the program what you expected based on the discussion above? Why or why not?

B.19: Try running the program with the matrix values in (B.11). Is the solution provided by the program what you expected based on the discussion above? Why or why not?

The final limitation has to do with whether there is even a solution or not. Certainly, if one or more of the equations is not independent then there is no single solution. You already showed that with two equations in section B.3. With more than two equations it can sometimes be difficult to see if all the equations are independent. For example, consider the following set.

$$\begin{bmatrix} 1 & 3 & -1 \\ 4 & -1 & 2 \\ 2 & -7 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \\ 0] \end{bmatrix} \tag{B.13}$$

At first glance these may appear to be independent since there is no single equation that is a multiple of any other equation. However, it turns out that you can get the third by taking the second and subtracting twice the first.

B.20: Try running the program with the matrix values in (B.13). Is the solution provided by the program what you expected? Why or why not?

B.5 The greenhouse effect

The **greenhouse effect** refers to how Earth's surface receives radiation not only from the sun but also the atmosphere, which means the surface is warmer than it would be without the atmosphere. To figure out what the surface temperature would be with an atmosphere, we need to solve a set of linear equations, which you can do using your program.

I won't show you a complete derivation of the set of equations but I do want to show you what the equations are and where they come from. Here are the assumptions we will make.

A) We'll treat the atmosphere as a single object with a single temperature. That is not realistic, of course, but it is a good starting point, and we'll address it later by splitting up the atmosphere into layers (the more layers, the more accurate this assumption).

B) We'll assume a certain amount of radiation from the sun, given by the solar constant (1361 W/m^2), and that 30% is reflected (off the surface and clouds and such) and thus will not be impacting the situation. These numbers are actually very close to what NASA says. Earth receives sunlight only on one side so we'll average this out over the entire surface by dividing by four since the surface area of a sphere is four times its cross-section area.

C) We'll assume the radiation from the sun (minus the reflected part) passes through the atmosphere without being absorbed at all. Given that most of the solar radiation is in the visible range and that we can see objects out in space (like the sun and moon), this seems like a reasonable assumption. In reality, about a third of the solar radiation is absorbed by the atmosphere. The equations can be written without this assumption but that makes the equations more complicated.

D) We recognize that the surface and atmosphere are at a temperature where they radiate in the infrared, which is absorbed much better by

the atmosphere than visible light. We'll assume that 90% of the infrared from the surface is absorbed by the atmosphere (very close to what NASA says). We'll assume that the surface absorbs 100% of the infrared that it receives from the atmosphere.

E) We'll assume that objects are as effective at emitting radiation as they are at absorbing it (Kirchhoff's law of thermal radiation). This means that Earth emits at a rate equal to σT^4 (where σ is the Stefan-Boltzmann constant of 5.67×10^{-8} W/m^{-2}K^{-4} and T is the temperature) while the atmosphere emits at $\epsilon\sigma T^4$ where ϵ is the **emissivity** (**absorptivity**) and equal to 90%.

F) We'll assume that the surface and the atmosphere both emit radiation in an amount that exactly balances how much they absorb. This is called **radiation balance**.

B.5.1 Radiation balance at Earth's surface

From assumption B, the surface is absorbing radiation from the sun at a rate equal to $S_0(1-a)/4$, where S_0 is the solar constant (1361 W/m^2) and a is the fraction reflected (30%; known as the **albedo**).

From assumption E, the surface is also absorbing radiation from the atmosphere at a rate equal to $\epsilon\sigma T^4_{\text{atmos}}$ but, at the same time, it is emitting at a rate equal to σT^4_{sfc}.

From assumption F, the amount emitted must balance the amount absorbed, leading to the following equation.

$$\sigma T^4_{\text{sfc}} = \frac{S_0}{4}(1-a) + \epsilon\sigma T^4_{\text{atmos}} \tag{B.14}$$

B.5.2 Radiation balance for the atmosphere

From assumption E, the atmosphere is absorbing radiation from the surface at a rate equal to $\epsilon\sigma T^4_{\text{sfc}}$ but, at the same time, it is emitting at a rate equal to $2\epsilon\sigma T^4_{\text{atmos}}$, the factor of 2 coming from the fact that it emits both downward (to Earth) and upward (to space).

From assumption F, the amount emitted must balance the amount absorbed, leading to the following equation.

$$2\epsilon\sigma T^4_{\text{atmos}} = \epsilon\sigma T^4_{\text{sfc}} \tag{B.15}$$

B.5.3 Greenhouse temperature in single-layer model

Equations (sfc.rad.balance.eq) and (atmos.rad.balance.eq) have two variables, T_{sfc} and T_{atmos}. We can solve for those two using the linear equation solver. Before we do, let's rewrite the two equations so they look similar in structure to what we had in section B.3.

First, we'll use x_0 and x_1 for σT_{sfc}^4 and $\sigma T_{\text{atmos}}^4$ (the surface radiation and atmospheric radiation), respectively. Then I'll rearrange the equations slightly to get the following:

$$\begin{gathered} -x_0 + \epsilon x_1 = -S_0(1-a)/4 \\ \epsilon x_0 - 2\epsilon x_1 = 0 \end{gathered} \tag{B.16}$$

In matrix form this would be:

$$\begin{bmatrix} -1 & \epsilon \\ \epsilon & -2\epsilon \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} -S_0(1-a)/4 \\ 0 \end{bmatrix} \tag{B.17}$$

B.21: Run your code with the values specified in (B.17), with $\epsilon = 0.1$ and the values of S_0 and a given above. What does your program predict is the equilibrium temprature of the surface and the atmosphere?

B.5.4 Extending to n atmospheric layers

Now extend the process to account for n layers. The surface still emits at x_0 and absorbs $S_0(1-a)/4$ but now it has n layers of atmosphere emitting radiation, each at ϵx_i down toward the surface. The higher up the layer, though, the more the radiation is absorbed by the layers beneath it. All of the emission from the bottom layer (ϵx_1) is absorbed at the surface but only $(1-\epsilon)$ of the radiation in the second-to-bottom layer is absorbed. That means the surface absorbs $(1-\epsilon)\epsilon x_2$ from the second-to-bottom layer. Each layer above gets an extra ϵ taken out as it goes through each layer so the total for the surface is as follows:

$$-\frac{S_0}{4}(1-a) = -x_0 + \epsilon x_1 + (1-\epsilon)\epsilon x_2 + (1-\epsilon)^2 \epsilon x_3 + \cdots + (1-\epsilon)^{n-1}\epsilon x_n$$

This can be written as follows:

$$-\frac{S_0}{4}(1-a) = -x_0 + \sum (i = 1^n \epsilon (1-\epsilon)^{i-1} x_i \tag{B.18}$$

Each layer of atmosphere still emits $2\epsilon x_i$ but how much it receives from the surface and layers below it depends how many layers the radiation must pass through, and we need to multiply by $(1-\epsilon)$ for each layer it passes through. The same is true for radiation from layers above it.

$$0 = \epsilon(1-\epsilon)^{i-1}x_0 - 2\epsilon x_i + \sum_{j=1}^{i-1}\epsilon^2(1-\epsilon)^{i-1-j}x_j + \sum_{j=i+1}^{n}\epsilon^2(1-\epsilon)^{j-i-1}x_j \quad \text{(B.19)}$$

In matrix notation, equations (B.18) and (B.19) look like the following:

$$\begin{bmatrix} -1 & \epsilon & (1-\epsilon)\epsilon & (1-\epsilon)^2\epsilon & \cdots & (1-\epsilon)^{n-1}\epsilon \\ \epsilon & -2\epsilon & \epsilon^2(1-\epsilon) & \epsilon^2(1-\epsilon)^2 & \cdots & \epsilon^2(1-\epsilon)^{n-2} \\ \epsilon(1-\epsilon) & \epsilon^2(1-\epsilon) & -2\epsilon & \epsilon^2(1-\epsilon) & \cdots & \epsilon^2(1-\epsilon)^{n-3} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ \epsilon(1-\epsilon)^{n-1} & \epsilon^2(1-\epsilon)^{n-2} & \epsilon^2(1-\epsilon)^{n-3} & \epsilon^2(1-\epsilon)^{n-4} & \cdots & -2\epsilon \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ . \\ . \\ . \\ x_k \end{bmatrix} = \begin{bmatrix} -S_0(1-a)/4 \\ 0 \\ 0 \\ . \\ . \\ . \\ 0 \end{bmatrix}$$

B.5.5 Running the modified code

The code B5:radiationbalance is essentially the same program you developed before in section B.4 but with the greenhouse equations described above.

B.22: When you run the program, it asks for the number of atmospheric layers. Does specifying more layers mean the atmosphere is thicker? If not, what does it mean?

B.23: Look up Earth's blackbody temperature (the equilibrium temperature with no atmosphere). Should we expect the program to give a surface temperature similar to Earth's blackbody temperature? If so, why? If not, why not? You might find it useful to run the program with zero, one, and two atmospheric layers before answering.

B.24: Run the program with 20 atmospheric layers. Does the predicted surface temperature roughly equal to the average surface temperature on Earth (288 K)? If not, why would that be?

B.5.6 Analyzing the modified code

It is useful to examine the code, not only to see how the greenhouse equations were incorporated into the code but also to examine some new Python features we haven't used before. I recommend you go back and forth between the code and what I describe below.

- The first line uses the `input` function, which provides a prompt allowing the user to input how many atmospheric layers they want. The result of that is passed to the `float` function, which converts the answer from a string of characters to a number.[vi]
- The next line sets the value of ϵ, which the program calls `eps`. The atmosphere as a whole absorbs 90% of the radiation emitted from the surface but that doesn't mean each layer does. After all, if the lowest layer absorbed 90% all by itself then the second layer would absorb 90% of the 10% that passed through, leaving only 1% to get to the third layer, and so on. Basically, the amount that would pass through the entire atmosphere would be $(1-\epsilon)^n$, where n is the number of atmospheric layers. This particular line of code calculates what ϵ must be such that $(1-\epsilon)^n = 10\%$ (for a total absorptivity of 90%).
- The next three lines assign the values of S_0, a, and ϵ and should be self-explanatory.
- The next three lines let the program know that there will be three arrays (matrices), which are called `A`, `B` and `val`. I use `val` to set up each row of `A`. The equations only use `A` and `B`, but using `val` helps to see what is going.

[v]Above that level, the air is rather thin and is impacted by the absorption of solar radiation by ozone (which our model doesn't include) and the very upper atmosphere (also not included in our model).

[vi]The term **float** refers to **floating point**, in comparison to **fixed point**, where the point between >1 and <1 is always in the same place within the string of numbers (so, for example, an 8-digit number like 12345678 would correspond to 1234.5678). In a sense, a floating point number is like a number in scientific notation.

- The next five lines set up the first row of `A` and `B`, to make it match equation (B.18). Notice how it uses the `append` function to add a value onto the array. Also notice how the program raises `(1-eps)` to the power of `i` whereas equation (B.18) has it raised to the power of `i-1` . This is because the `range(N)` function goes from `0` to `N-1` so python has already subtracted one for us.
- The next set of lines sets up the rest of the rows of `A` and `B`, to match equation (B.19). You should see how it matches what is in equation (B.19). Notice how it "reinitializes" the `val` array each time. This is because we want to reset the values of `val` with new values.
- Except for the last four lines, the rest of the code matches what we had before. The last four lines figure out the temperatures that correspond to the values of x_0 and x_1. Remember that x_0 and x_1 are not the temperatures. To get the temperatures, it needs a loop that solves $x_i = \sigma T_i^4$ for the temperatures.

At this point, you should be able to see that the core of the code is pretty much the same as what we had before, just applied specifically to the greenhouse effect problem.

B.25: If the program is modified to use a total emissivity equal to 75% instead of 90%, the surface temperature ends up matching the average surface temperature on Earth (288 K). Describe the change you'd need to make to the program to use a total emissivity equal to 75% instead of 90%.

Problems

Problem B.1: Suppose the code used the following values for the A and B matrices.

```
A = [ [1., -3., 1.], [2., -8., 8.], [-6., 3., -15.]  ]
B = [4., -2., 9.]
```

In the python code, what is the value of `A[1][2]`?
a. -3
b. 2
c. 8
d. 3
e. None of the above; provide the actual answer

Problem B.2: When solving for the temperature profile of the atmosphere due to the greenhouse effect, what did x_1 refer to?
a. The temperature at Earth's surface
b. The temperature at the lowest layer of the atmosphere (just above the surface)
c. The temperature at the highest layer of the atmosphere
d. The temperature of the sun
e. None of the above; provide the actual answer

Problem B.3: When the program is initialized with the following lines,

```
A = [ [2., -1., -1., 2.], [-1., 1., 2., -2.],\
      [0., 1., 1., 1.], [3., 2., 1., 4.]  ]
B = [-2., 5., 0., -10.]
```

it produces the following output for B.

```
[-2.125, -5.875, 5.125, 0.75]
```

In comparison, when the program is initialized with the following lines,

```
A = [ [0., 1., 1., 1.], [-1., 1., 2., -2.],\
      [2., -1., -1., 2.], [3., 2., 1., 4.]  ]
B = [0., 5., -2., -10.]
```

it produces the following output for B.

```
[NaN, NaN, NaN, NaN]
```

Compare the second set of equations with the first set. What is it about the first set that produces the error, and why would that produce an error?

Problem B.4: Suppose you are given the following set of equations.

$$w + x + y + z = -2$$
$$2w + x - y - 2z = 5$$
$$-w - x + 2y + 2z = 0$$
$$w + 2x + 3y + 4z = -10$$

When the program is used with these values, the first sequence of steps (what the readings refer to as step S1) produces the following values for arrays A and B.

```
A = [[1, 1, 1, 1], [0, -1, -3, -4], [0, 0, 3, 3], [0, 0, 0,
0]]
B = [-2, 9, -2, 0.333333]
```

However, after completing step S2 and finishing, it prints out the following for B.

```
B = [Infinity, -Infinity, -Infinity, Infinity]
```

What was it about the original four equations that led to the error? Be specific.

Problem B.5: Our code has the following lines for step S1.

```
for k in range(N-1):
  for i in range(k+1,N):
    factor = A[i][k] / A[k][k]
    B[i] = B[i] - B[k] * factor
    for j in range(k,N):
      A[i][j] = A[i][j] - A[k][j] * factor
```

Suppose we replaced the `j` loop with the following. Would the code still work? Why or why not?

```
    for j in range(N):
```

C. Non-Linear Equations

- Physics context: Lagrange point
- Programming skills: while loops, +1 notation, if/else
- Computation skills: Search algorithms

C.1 Introduction

In chapter B, we reviewed how to solve a single linear equation with one unknown. While a single linear equation, with one unknown, relatively simple to solve, either by hand or computer, a single equation with one unknown that is raised to various powers (like $x^2 + 2x = 3$) or involves other functions (like $x^2 + \sin x = 3$) can be quite a bit more complicated to solve. In this part, you'll look at how to solve such equations computationally. Since this is a physics class, not a computer class, we'll use a physics situation that involves an equation of one unknown that is complicated enough to be difficult (or impossible) to solve without a computer. At the same time, it needs to be simple enough that it can be understood by someone who has only taken the first semester or two of college physics.

With those points in mind, the context for this part is to identify a **Lagrange point** (or Lagrangian point).[i] A Lagrange point occurs where an object's orbital period around the sun equals that of Earth's orbital period around the Sun. That means it would appear to be stationary with respect to the Sun when viewed from Earth.

There are several Lagrange points. We will focus on the one that lies between the Sun and Earth. The following equation gives that Lagrange point location:

$$(\alpha^3 - 1)(\alpha - 1)^2 + \beta\alpha^2 = 0 \tag{C.1}$$

In the expression, α is the location of the Lagrange point in terms of a fraction of the Earth-Sun distance – this is the quantity we'll eventually determine –

[i] For more information, see https://epic.gsfc.nasa.gov/.

and β is the ratio of the Earth to Sun mass.

$$\alpha = \frac{r}{r_e}$$
$$\beta = \frac{m_e}{m_s}$$

The equation is very hard to solve analytically[ii] so we'll use the computer to "search" for the answer. There are many ways to do this, of which we will examine five. We'll examine the simplest method in section C.1 and then progress through more and more sophisticated methods in sections C.2-C.5.

C.1.1 Deriving the expression

Before examining how we can solve for the location of the Lagrange point, it helps to first examine how we got the equation in the first place, so you can better interpret the results.

To derive the relationship, you need to first recognize that the Lagrange point is where the two *gravitational forces* acting on the object – that due to Earth and that due to the Sun – add up to the *centripetal force* needed to accelerate the object around the Sun with an orbital period equal to that of Earth's orbital period around the Sun. The gravitational force due to the Sun (of mass m_s) on an object of mass m a distance r away is:

$$G\frac{mm_s}{r^2}$$

The gravitational force due to Earth (of mass m_e) on an object of mass m a distance r away from the Sun is:

$$G\frac{mm_e}{(r_e - r)^2}$$

The two forces above (acting in opposite directions) must equal the product of the object's mass and its acceleration (from Newton's second law). Since the object is moving in a (somewhat uniform) circle, the acceleration must be

[ii] In other words, I don't know how to do it, although we can get an approximate answer that is very close to the actual.

toward the center of the center and have a magnitude equal to the following, where the circle has radius r and period T:[iii]

$$\frac{4\pi^2 r}{T^2}$$

To simplify the resulting expression, do the same process for Earth. In other words, find the gravitational force on Earth due to the Sun and set that equal to Earth's mass times its acceleration, keeping in mind that Earth must have the same period as the object at the Lagrange point.

C.1: Use the expressions above (gravity, centripetal force) along with Newton's second law and the definitions of α and β to derive the equation that gives the Lagrange point between the Sun and Earth (equation C.1). Hint: construct two equations, one for an object at the Lagrange point and one for Earth, then combine. Also note that an object at the Lagrange point has the same orbital period as Earth.

C.1.2 Setting up the equation

In equation (C.1), α is the fraction of the sun-Earth distance where the Lagrange point is. We basically want to know the values of α (there may be more than one) that make the expression true. The solutions are called the **roots** of the expression, and so we are basically going to do some **root finding**.

C.2: For our purposes, it is sufficient[iv] to use 3×10^{-6} for β. Calculate the value of the left side of equation (C.1) for the following values of α: 0, 0.5 and 1. Based on your results, is the root one of the three values: 0, 0.5 or 1? If so, which one and why? If not, why not?

Before showing a way to find the root computationally, I first want to comment on how equation (C.1) is arranged. There are lots of ways to arrange the expression, and there is no "right" way to do so. However, there are some ways that are better than others if we want to solve for α . What is nice about the arrangement in equation (C.1), for example, is that α doesn't

[iii] This is often written as v^2/r but since $v = 2\pi r/T$, this can also be written as $4\pi^2 r/T^2$, which is better suited for this particular situation.

[iv] Earth's mass is roughly 5.97×10^{24} kg while the Sun's mass is roughly 1.99×10^{30} kg.

appear in a denominator anywhere. For example, consider if we wrote it the following way:

$$\alpha^3 = 1 - \frac{\beta\alpha^2}{(\alpha - 1)^2} \tag{C.2}$$

While equation (C.2) is equivalent to equation (C.1), using equation (C.2) runs the risk of dividing by zero if we choose to try $\alpha = 1$. While *we* know that $\alpha = 1$ is not a solution (that would put the Lagrange point right at Earth), the *computer* doesn't know this, and we don't want to run the risk of a divide by zero error.

Another advantage of equation (C.1) is that the α dependence is all on one side (the left side). Again, not crucial, but it allows us to focus our attention on one side. Basically, we have just a single function of α, which we can call $f(\alpha)$, which is equal to zero, and our task is to find the **zeroes** of the function $f(\alpha)$:

$$f(\alpha) = (\alpha^3 - 1)(\alpha - 1)^2 + \beta\alpha^2 \tag{C.3}$$

By the way, for the computer to find the root of a function, the function itself does not need to equal zero, as in this case. However, it turns out to be is easier to keep track of things when solving for the zeroes of a function, as with equation (C.1).

C.3: Does the right side of equation (C.3) necessarily equal zero at the Lagrange point location? Why or why not?

C.1.3 Exploring the properties of the function

Before solving for the root of any function, it is helpful to get a sense of where the roots are likely to be. You don't need to know exactly where the roots are (after all, if you did, there would be no reason to solve for them) but it does help if you know roughly where they are, as this can guide which computational approach is best for a given situation.

In this case, the function is given by equation (C.3) and the Lagrange points are wherever $f(\alpha)$ is equal to zero, as expressed in equation (C.1). Plugging in zero for α gives a value of $f(\alpha)$ that is negative, while plugging in 1 for α gives a value of $f(\alpha)$ that is positive. In other words, $f(0) < 0$ and $f(1) > 0$. That means α must be between 0 and 1, which makes sense – the

Lagrange point must be between the sun and Earth (assuming the function is continuous).

C.4: It was mentioned that if the function is continuous and $f(0) < 0$ and $f(1) > 0$ then α must be between 0 and 1. Why does the function need to be continuous?

C.5: Given your results in question C.2 (examine the values of the function at 0, 0.5, and 1), where do you expect the root to be, between 0 and 0.5 or between 0.5 and 1, and why?

C.1.4 The brute force method

With the brute force method, we start at one boundary (let's say $\alpha = 0$) and calculate the value of the function there. Then we change α by a little bit (which I'll call $\delta\alpha$) and calculate the value of the function at the new value of α. If the value of the function is positive for one and negative for the other then we know that the solution is somewhere in between the two α values.

A brute force Python program for finding the Lagrange point is CLagrange-bruteforce.

Try out the program. You should find that the Lagrange point is between 0.9900 and 0.9901. Examine the code. You should be able to see that the program increments α by an amount equal to 10^{-4}, starting from zero, and continues to increase α by 10^{-4} until it finds that the value of $f(\alpha)$ switches signs.

There are two new Python statements that we haven't seen before, and they are in the following three lines:

```
while f(a)*f(a+res)>=0:
  a = a + res
  nint += 1
```

The first new statement is the `while` statement. This tells the program to carry out the indented statements that follow it over and over until, in this case, `f(a)*f(a+res)` becomes greater than or equal to zero. Basically, if the zero lies between `f(a)` and `f(a+res)` then their product would be negative (one will be negative and one will be positive).

The second new statement is the ''+='' notation, which is short-hand for telling Python to add the quantity to its current value. In this case, `nint += 1` is equivalent to `nint = nint + 1`.

Based on the code, answer the following questions.

C.6: Describe how the program knows when to stop searching for a solution. Do not simply provide the code or the line number.

C.7: Describe how the program can determine the number of digits to include in the numbers it prints out. Do not simply provide the code or the line numbers. Note: Without these lines, the python code won't print out enough digits for our purposes.

You should notice that the code has the following two lines that allow it to determine the value of $f(\alpha)$.

```
def f(alpha):
  return (alpha**3-1)*(alpha-1)**2+beta*alpha**2
```

Basically, the code uses the `def` statement to define a function named `f`. The input to `f` is a parameter called `alpha`, which it then uses in the indented line that follows.

C.8: Copy the code into your own account and name it something like "brute-force". Then, in the two lines in the code that match the two lines shown above for the definition of `f`, replace each instance of `alpha` with `x`. Does that impact the code in any way? If so, why? If not, why not?

C.2 Improving the method

Edit your code to search for the solution with a resolution of 10^{-6} (instead of 10^{-4}).

You should find that it takes a lot longer, which is a problem with the brute force method when, as in this case, the root is not close to our initial guess and we want a very small resolution. It takes even longer to come to a solution when the resolution is decreased to 10^{-7} (i.e., 10 times longer). What if we want the resolution be 10^{-9} (i.e., 1000 times longer)?

To quicken the search, we'll modify the program. We'll examine two modifications in this section, compare the two in section C.3, then introduce more

sophisticated methods in sections C.4 and C.5. There are additional methods that we won't get to. However, you'll learn the general approach and some of the issues that one can run into.

C.2.1 Modified brute force method

With the modified brute force method, we start searching as before (with the brute force method) but with some large resolution like 10^{-1}. We increment α as before, but when the sign of $f(\alpha)$ changes, rather than stopping there, we decrease $\delta\alpha$ by a factor of 10 and then increment by the new $\delta\alpha$ until we once again pass the solution, at which point we again decrease $\delta\alpha$. In this way, we continue to search with finer and finer increments until the increment $\delta\alpha$ goes below some threshold (like 10^{-9}).

To make this modification, you can either modify your brute force code or you can copy the brute force code to make a new program with a name like "modifiedbruteforce". Rather than stopping our search when the sign of $f(\alpha)$ changes, we instead want to stop our search when the search increment gets smaller than our target resolution. That means that we need to replace the `while` statement with a different one.

This can be done by first setting up a search increment, which we can call `inc`, that you can initially set to 0.1.

```
inc = 0.1 # starting increment
```

Then, rather than search while `f(a)*f(a+res)` is positive, we instead want to search while the search increment is bigger than `res` (our target resolution), so replace the while statement with the following.

```
while abs(inc)>=res:
```

Inside the while loop, we only want to change `a` if the sign of $f(\alpha)$ doesn't change. Otherwise, we want to decrease the search increment. This can be done by using an `if/else` check instead of the two lines used before:

```
  if f(a)*f(a+inc)<0:
    inc = inc/10.
  else:
    a += inc
    nint += 1
```

C.9: By how much does the code decrease the search increment when it finds a region with a root in it?

C.10: Modify your program to use the modified brute force method, stopping when you find the solution within 10^{-9}. Previously (with the brute force method) it was shown that the solution is somewhere between 0.99 and 0.9901. What are the bounds now?

C.2.2 Bisect method

The brute force method is faster but it can still take a while if the root is far from the initial guess. To fix this problem, rather than starting with a single first guess of α and moving progressively in one direction to find the zero, the bisect method starts with *two* choices of α, which I'll designate as α_0 and α_1, where we know one value of $f(\alpha)$ is positive and the other is negative. This ensures that the solution is somewhere between α_0 and α_1.

For our next guess, we consider the value of α that is *midway* between α_0 and α_1, which I'll indicate as $\alpha_{1/2}$. If $f(\alpha_{1/2})$ has the same sign as $f(\alpha_0)$ then we know the root must be between $\alpha_{1/2}$ and α_1. Otherwise, it must be between α_0 and $\alpha_{1/2}$. Either way, we now have a range that is half the size of what it was before.

Continuing, we then consider a new value of α that is midway between the two new points, and on and on, continually halving the region of interest until the region gets smaller than the desired threshold. At that point, we know we've found the solution to a resolution smaller than that threshold.

In terms of the computer code, the structure is like the modified brute force method except for three things.

- Rather than stopping our search when the search *increment* gets smaller than our target resolution, we instead want to stop the search when the search *range* gets smaller than our target resolution. That means that you need to modify the `while` statement to reflect this. For example, if you use `a0` and `a1` to represent the two values α_0 and α_1 then the `while` statement would check for when `abs(a1-a0)` reaches the target resolution.
- You no longer need to calculate the increment but you do need to calculate the midpoint value, $\alpha_{1/2}$. For example, if you use `amid` to

represent the midpoint value then you'll need a statement like `amid = 0.5*(a0+a1)` whenever you want to calculate the midpoint value.

- The inside of the while loop also needs to be changed. Rather than incrementing `a`, you instead want to either shift `a0` (the left side of the search area) to `amid` or shift `a1` (the right side of the search area) to `amid`, depending on which side the root happens to be in. If the product of $f(\alpha_0)$ and $f(\alpha_{1/2})$ is negative then you know the solution is somewhere in that range and you can reset `a1` to be `amid`. Otherwise, it is in the range of $f(\alpha_{1/2})$ and $f(\alpha_1)$ and you can reset `a0` to be `amid`.

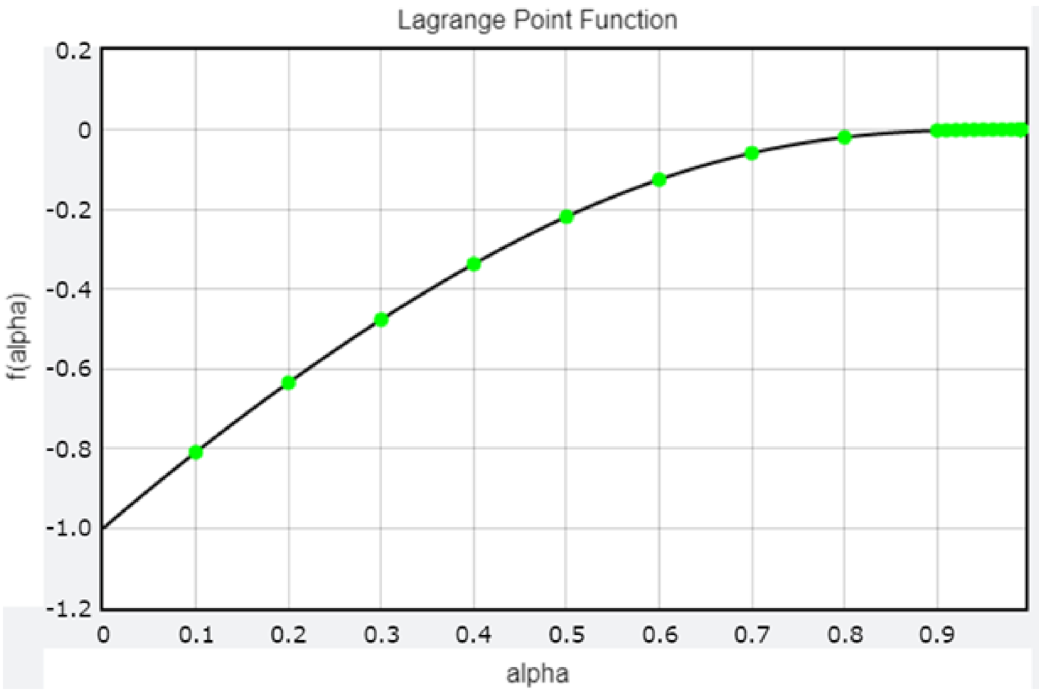
C.11: Add the bisect method to your code or copy the modified brute force code to a new program (name it something like "bisect") and modify it to use the bisect method. Start with 0 and 1 as your initial guesses (since we know it needs to be between the sun and Earth). Continue "bisecting" the region of interest until the region size gets smaller than 10^{-9}. What are the bounds of the root now?

C.12: Compare the number of iterations needed for the bisect method with the modified brute force method. Which takes less iterations?

C.3 Comparing the modified brute force and bisect method

With the two techniques used so far, you should have found that the solution is close to 0.9900296712 (for a resolution of 10^{-10}). To get a better sense of what is going on, let's compare the modified brute force method and the bisect method. A good way to do this is to first graph what is going on.

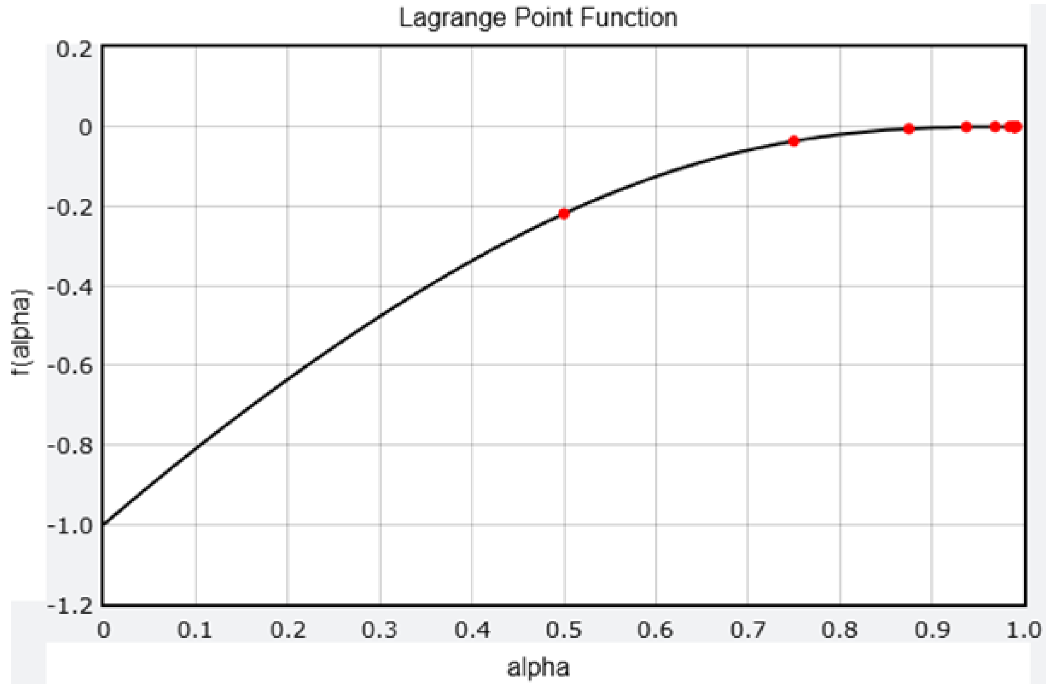
The following plot shows the progression during the Modified Brute Force method. Each ? trial is shown by a green dot. Basically, the program starts at the left side (at ?=0) and progresses steadily toward the right, first with 0.1 spacing then 0.01 spacing then 0.001 spacing and so on.



Since we started at zero, the number of iterations needed for the modified brute force method is the sum of the value of each digit: $9+9+0+0+2+9+6+7+1+2=45$. The more 9's in the solution, the more steps it needs to get to the answer. Assuming, on average, a digit value of 5, the number of steps required for the brute force method should be around $5m$, where m is the number of digits.

C.13: What assumption is made when asserting that the number of steps is $5m$ for the brute force method? Under what conditions would it be less than this, if any? Under what conditions would be greater than this, if any?

In comparison, the following plot shows the progression for the bisect method. Again, each α trial is shown by a dot, this time in red. The program starts with two points, at $\alpha = 0$ and $\alpha = 1$ (not plotted) and then identifies the 0.5, 0.75, 0.825 and so on, continually cutting the range in half each time.



Since the bisect method necessarily cuts the increment size in half every step, the increment size is $(1/2)^n$ after n iterations. It stops when the increment size gets to 10^{-9}, which means:

$$\left(\frac{1}{2}\right)^n = 10^{-9}$$

Solving for n:

$$n \log\left(\tfrac{1}{2}\right) = -9$$
$$n = \tfrac{9}{\log(2)} = 30$$

This is the number of iterations needed to get to a resolution of 10^{-9}. To get a resolution of 10^{-m}, the bisect method would require $m/\log(2)$ steps.

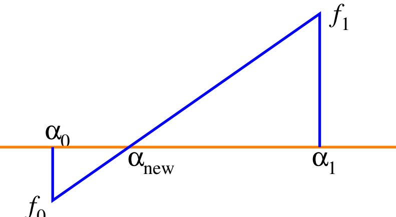
C.14: How do the predicted number of steps for the two methods (see above) compare to your actual number of steps? For each, why does the actual match or not match the prediction?

C.15: The bisect method is faster for the Lagrange location problem. Are there any situations (with a different root value) when the modified brute force method should be better than the bisect method (i.e., requires less steps)? If so, when? If not, why not?

C.4 Secant method

Rather than choose the value of $f(\alpha)$ at the MIDDLE of a range to determine the next range to consider (as we do with the bisect method), with the secant method we use the two values of $f(\alpha)$ at our range boundaries to figure out where the solution is likely to be and use that for the next guess. Basically, we'll assume that if $f(\alpha)$ is closer to zero at one end then the other, then the solution must be closer to that end and we'll narrow down the range to be closer to that side, rather than just cutting the range in half each time. For example, suppose you have two α's: α_1 and α_2. If $f(\alpha_1)$ is closer to zero than $f(\alpha_2)$ is to zero then the solution should be closer to α_1 and so we should be able to select a new range that is closer to α_1.

To show how we do this, consider the figure, where the orange horizontal line corresponds to $f(\alpha) = 0$. Suppose we have our search range from α_0 to α_1, and the function at α_0 (indicated as f_0) is closer to zero than the function at α_1 (indicated by f_1).



Since f_0 is closer to zero than f_1 is, that implies that the solution is closer to α_0 than α_1. Consequently, rather than splitting the domain directly in half and using the lower half as our next search domain, we instead set our new search domain from α_0 to α_{new}, where α_{new} is shifted toward α_0 via a secant line between f_0 and f_1 as shown (which is why it is called the **secant method**).

Mathematically, consider the base and height of the two similar triangles:

$$\frac{\alpha_{\text{new}} - \alpha_0}{f_0} = -\frac{\alpha_1 - \alpha_{\text{new}}}{f_1} \tag{C.4}$$

We can solve this for α_{new}:

$$\alpha_{\text{new}} = \frac{(\alpha_0 f_1 - \alpha_1 f_0)}{f_1 - f_0} \tag{C.5}$$

Note: One of the nice things about this secant technique is that the solution doesn't need to be between α_0 and α_1 (so both f's could be positive or both negative), as this method will simply "extrapolate" to see where the next search region should be. Unlike the bisect method, it is not restricted to the region between the two guesses.

C.16: Show, via specific algebraic steps, how equation (C.5) is obtained from equation (C.4).

The basic structure of secant method code is like that of the bisect method but instead of determining the midpoint of the range, you use equation (C.5), and that will automatically be used for the next range. Sample code is as follows.

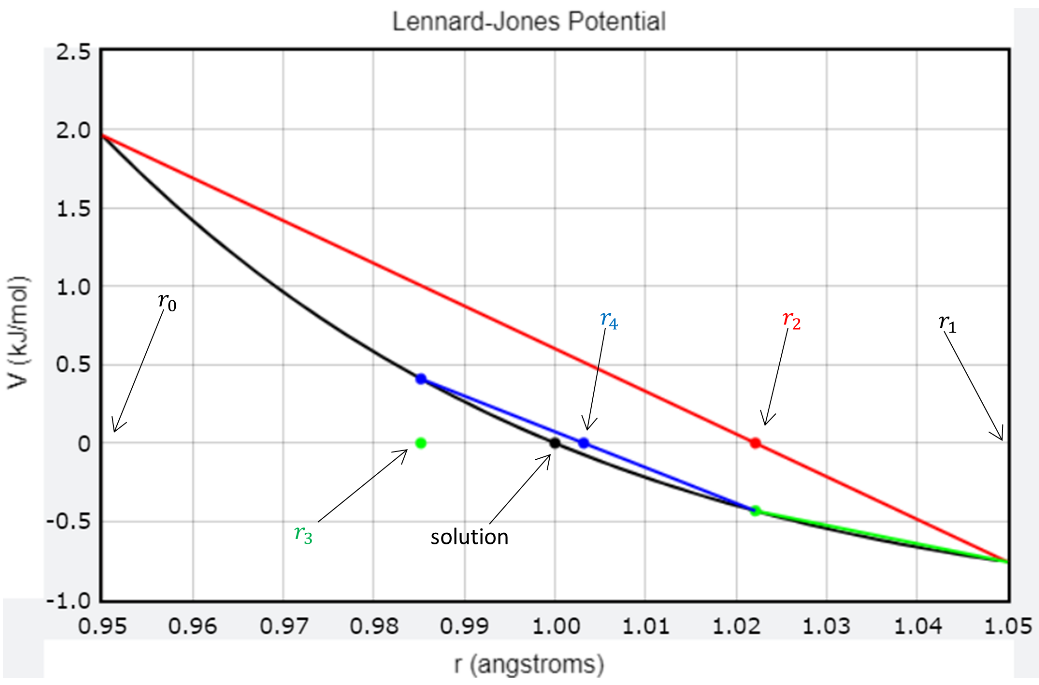
```
anew = (a0*f(a1)-a1*f(a0))/(f(a1)-f(a0))
a0 = a1
a1 = anew
```

If you are doing this, make sure that your `while` loop contains the check for `nint` as well as for the resolution. The secant won't necessarily converge onto the answer and you don't want it to get stuck in an endless loop. Sample code is as follows.

```
while abs(a1-a0)>res and nint<200:
```

C.17: Add the secant method to your code or copy the bisect code to a new program (name it something like "secant") and modify it to use the secant method. Start with 0 and 1 as your initial guesses and have it stop when the region size gets smaller than 10^{-9} or it reaches 200 iterations, whichever comes first. How does the secant approach compare to the modified brute force method and the bisect method?

You should find that the secant method requires more than 100 steps to get to a solution when the initial search region is bounded by 0 and 1. This is because the secant technique doesn't work well when the function is far from linear. To see why, consider the case for the Lennard-Jones potential, which describes the interaction between two atoms. It is non-linear, like our Lagrange equation. The basic physics is that atoms repel if they are too close together (as the positively charged nuclei repel) and they attract when farther apart (like two polarized objects). The dependence on distance is much stronger when closer than far apart, which leads to a non-linear dependence on distance, as seen by the black curve in the graph below.



Generally, we want to know where the minimum of the potential is (as that is the equilibrium position). However, let's suppose we want to know where the potential is zero. Granted, this is an arbitrary reference point, but it serves to illustrate a potential issue with the secant method.

From the graph, we can see that the function equals zero is at 1.00 Å, as indicated by the black dot. Let's see how the secant method searches for that value.

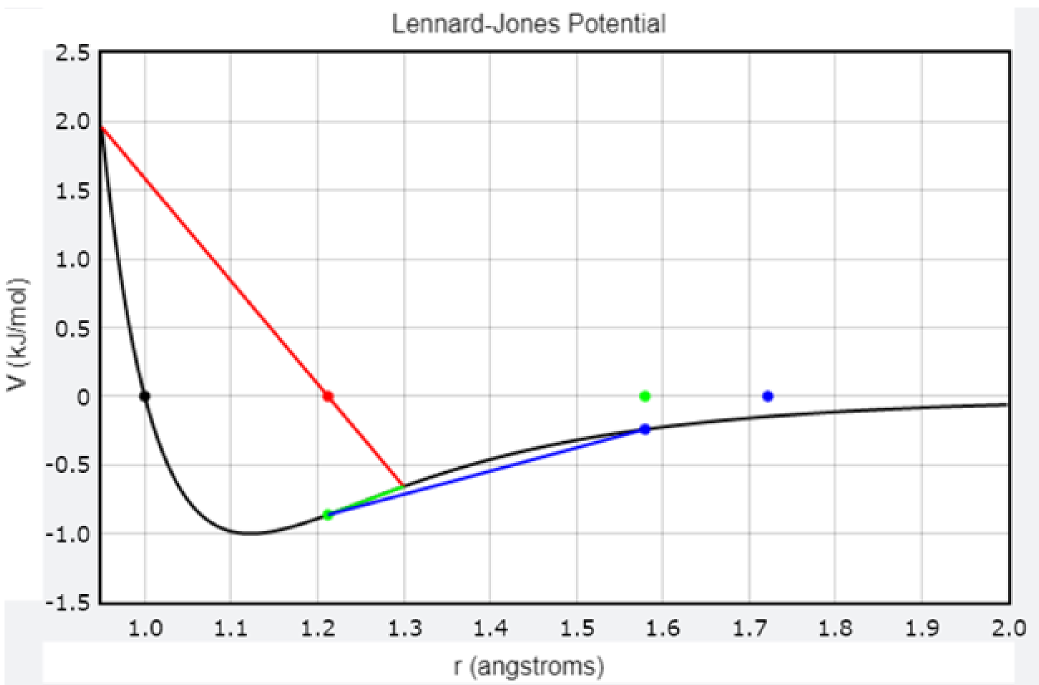
We'll start with our initial guesses being 0.95 Å(r_0) and 1.05 Å(r_1), as shown on the graph, where r is used to represent each guess (instead of α) since this function depends on r. The potential at 1.05 Å(around -0.75 kJ/mol) is closer to zero than the potential at 0.95 Å(around 1.9 kJ/mol), so the secant method figures that the solution should be closer to 1.05 Å. Indeed, when we draw a straight line between the two end points (red line), we see that the line crosses the axis at around 1.022 Å(see red dot with label r_2).

The new search region, then, is between 1.022 Å(r_2) and 1.05 Å(r_1). Even though the solution (block dot) is not within that region, the next secant line (green line) successfully points to a solution at 0.985 Å(r_3), which is closer to the solution than the previous guess of 1.022 Å.

The method then uses the search region between 0.985 Å(r_3) and 1.022 Å(r_2), and the new secant line (blue line) points to a solution at 1.003 Å(r_4), which is even closer to the solution. Continuing the process gets us closer and closer to the solution (not shown).

Note: Since the solution is exactly in the middle of our region, the bisect method would point us to the solution in just one step. That doesn't mean the bisect method is better. After all, in practice we *don't* know where the solution is. That is why we need to use numerical techniques. We could just as well argue that the secant method would point us to the solution in one step if the relationship was linear. However, if the relationship was linear, we wouldn't need to use numerical techniques.

So far, it looks like the secant method works, right? It turns out it only works in this case because the function is "relatively close" to linear in the range bounded by our initial guesses. Increase the range and the non-linearity becomes more apparent, and that can negatively impact the secant approach.[v] To illustrate, let's consider the same curve as before but expand the region, as shown below.



[v]In general, any curved line will appear straight if the portion is small enough.

What is being plotted is the same Lennard-Jones potential, as before. With the expanded range, we can clearly see that the two atoms are at the lowest potential when they are separated by about 1.12 Å. When closer than that or further than that, they are forced back toward 1.12 Å.

We'll now use the same "left" end as before (0.95 Å, with label r_0) but set the "right" end to 1.3 Å(r_1). As before, the secant method determines the next guess by "drawing" the straight line (red line) between the potential values at the ends. This gives the next guess at around 1.21 Å(r_2).

Notice how the new search region, between 1.21 Å(r_2) and 1.3 Å(r_1), is not only outside the solution but has a slope (green line) that points to a solution even further away (1.58 Å; r_3). Continuing with a search region between 1.21 Å(r_2) and 1.58 Å(r_3) leads to an even worse solution (1.72 Å; r_4). Basically, the method is searching for where the function goes to zero on the right (which is at infinity) and so continuing the process will just get us further and further away from the solution (not shown).

C.18: Why is the secant method unable to solve the Lagrange equation in less than 100 steps when the initial two points are 0 and 1? You may need to look at what is happening at each step (use some print statements).

C.19: In comparison to using 0 and 1 as the initial two points, the secant method can solve the Lagrange equation in less than 100 steps when the initial two points are 0.0 and 0.5 (and comes to a solution more quickly than either the modified brute force method or the bisect method). Why?

C.20: Lest you think that the secant method always works better when the range is small, or when the range is closer to the solution, compare the results for the Lagrange point search when the initial two points are 0.499 and 0.5 vs. 0.999 and 1. How many iterations does each take to get within 10^{-9} of the solution?

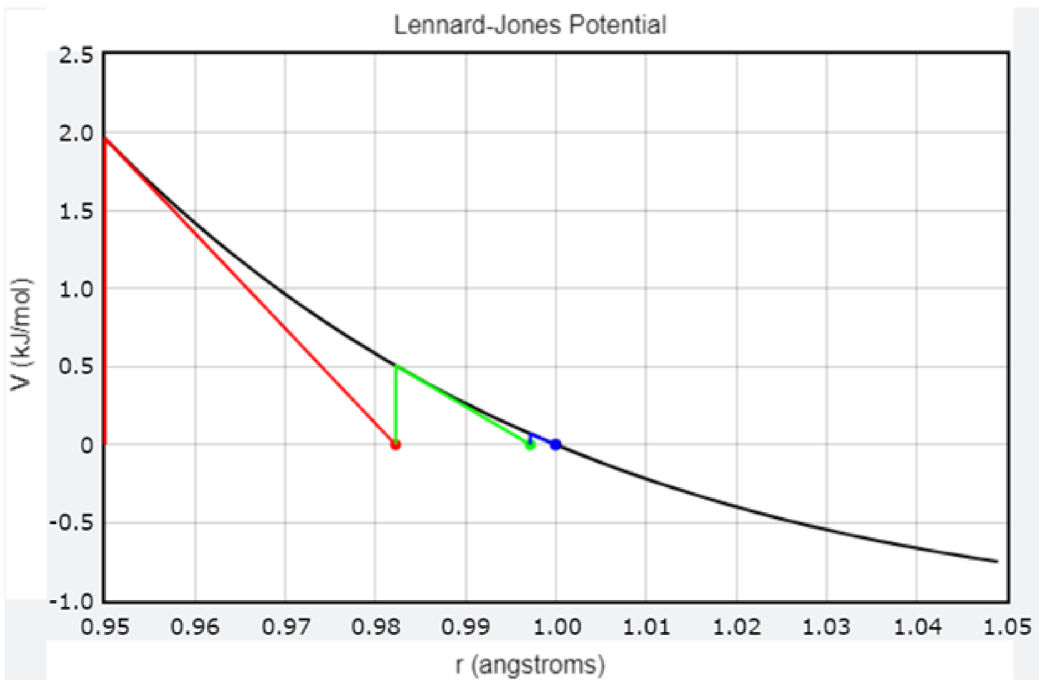
C.5 Newton's [aka Newton-Raphson] method

As discussed in section C.4, the secant method can find the root quickly but it depends a great deal on the selection of the first two guesses, particularly if the function is somewhat flat (as it is around $\alpha = 1$ in our case) or has multiple peaks and valleys. For example, some guesses can come to a solution prematurely, while other guesses end up getting caught in an endless

loop. For our situation, it turns out that the method works when the initial boundaries are 0.90 and 0.91 or 0.50 and 0.51 but it isn't clear at first why we would even try those.

To fix this problem, we'll explore a method that uses the *slope* of the function $f(\alpha)$ itself rather than the *average slope* between two points of $f(\alpha)$. This method is called **Newton's method** (or **Newton-Raphson method**).

To illustrate Newton's method, let's again consider the Lennard-Jones potential, illustrated by the black curve in the following graph. The solution, as before, is at $r = 1.00$.



With Newton's method, we choose a single point as our first guess. Here I have chosen to start at the left side of the domain (r_1 in the figure). We then calculate both the value of the function at that point, indicated by the vertical red line, and the derivative at that point, indicated by the slanted red line. The new guess is located along the base of the triangle created by the two lines.

Since the value of the derivative $f'(r_1)$ is equal to the height of the triangle,

$-f(r_1)$, divided by the width of the triangle, $r_2 - r_1$, we have:

$$f'(r_1) = \frac{-f(r_1)}{r_2 - r_1}$$

Solving for the new guess, we have:

$$r_2 = r_1 - \frac{f(r_1)}{f'(r_1)}$$

We then do the same at this new point, calculating the value of the function and derivative at r_2. This gives us the green triangle and leads us to the next guess at r_3. Continuing this process, we get the blue triangle and the guess at r_4.

The general solution is (using the notation for Lagrange equation):

$$\alpha_{n+1} = \alpha_n - \frac{f(\alpha_n)}{f'(\alpha_n)}$$

This can be very efficient if there are not multiple solutions within our domain. However, it requires knowledge of the derivative.

Fortunately, we know from chapter A that we can obtain the value of the derivative numerically[vi] – we simply need to use one of the techniques identified in section A.4. The second-order approach, for example, is:

$$f'(x) \approx \frac{f(x + \delta x) - f(x - \delta x)}{2\delta x}$$

Create a program to determine the Lagrange location via Newton's method. To do this, you'll need to create a second definition, one for the derivative of the function (you already have a definition for the function itself). The first line of the definition would be:

```
def fprime(alpha):
```

The second line of the definition, indented, would be the derivative of the Lagrange function (see, for example, the second-order approach above; you need to set a sufficiently small $\delta\alpha$ for the derivative calculation).

[vi] The Lagrange equation is just a polynomial equation and thus it is relatively easy to obtain the derivative analytically (using the power and product rules from calculus): $f'(\alpha) = (5\alpha^3 - 3\alpha^2 - 2)(\alpha - 1) + 2\beta\alpha$. However, it isn't always so easy.

You also need to change the line where you calculate the next guess. For the Newton method, the line should look like the following:

```
    anew = a1 - f(a1)/fprime(a1)
```

The overall structure should be similar to the secant method. However, if you are comparing `abs(a1-a0)` to the target resolution, keep in mind that the Newton method only uses one initial guess, not two. If you want, you can set `a0` to initially be something way off, like `-999`, as the program will only use it to check whether `abs(a1-a0)` is initially within the target resolution. After that, it will be replaced by `a1`, with `a1` being replaced by anew.

C.21: Create a program to determine the Lagrange location via Newton's method. Start with a guess at $\alpha = 0.5$ (middle of domain). How does the efficiency (number of steps) compare to what you obtained with the other methods?

C.22: Does the efficiency (number of steps) depend upon where you start within the domain? Try starting at the left end of the domain ($\alpha = 0$) and at the right end ($\alpha = 1$) rather than the middle ($\alpha = 0.5$). Why do you think this is?

Problems

Problem C.1: Suppose we want to find the stable Lagrange point on the other side of Earth (farther from the Sun) rather than the stable Lagrange point between Earth and the Sun. Would the expression for $f(\alpha)$ change from what we used? If so, how? If not, why not?

Problem C.2: When using the secant method for our Lagrange point problem:
(a) Why might it be better to start with the first two points being 0.5 and 0.6 rather than 0 and 1?
(b) Is it better to start with 0.4 and 0.5 rather than 0.5 and 0.6? Why or why not?
(c) Is it better to start with 1 and 0 rather than 0 and 1? Why or why not?

Problem C.3: Suppose we want to solve a linear equation.
(a) Which of the four numerical techniques (modified brute force, bisect, secant or Newton's) would you recommend for finding the solution? Provide your rationale.

(b) For the technique you've identified, does it matter which point or points you use for your initial guesses? Why or why not?

Problem C.4: Suppose you are given a nonlinear algebraic expression of unknown x and are told that there is only one zero in the domain of interest but there are multiple inflection points where the slope is zero. Which method would you suggest for finding the root of the expression (the value of x where the expression is zero) and why?

Problem C.5: A student recognizes that the Lagrange point is much closer to Earth ($\alpha = 1$) than to the Sun ($\alpha = 0$) so they create the following code to use an initial guess of 1 rather than zero (where `a` represents α).

```
inc = -0.1
a=1

while abs(inc)>10**(-9):
  if f(a)*f(a+inc)<0:
    inc = -inc/10
  else:
    a = a - inc
  nint += 1
```

(a) Which numerical method is being attempted?
(b) Are there any errors in the code? If so, how would you fix it (other than reverting to the same code we've used)?

D. Fourier Analysis

- Physics context: Data analysis
- Programming skills: Graphs, reading files
- Computational skills: Dealing with noise
- Mathematics skills: Fourier analysis, correlation coefficients

D.1 Reading data files and making graphs

D.1.1 The context

Many times we are given a series of measurements or observations and wish to know if there is a "signal" embedded in the data. An example of a series of measurements would be the hourly air pressure values over the course of a week or month (if your phone can measure pressure, you could download a barometer app and do this yourself). An embedded "signal" would be the semidiurnal (twice per day) tidal oscillation that is present in the air pressure measurements due to the sun warming up the air during the day.[i]

In any event, in this part, we'll examine how the computer can identify whether there are any sinusoidal patterns within a series of data and, if so, what the periods (frequency) and amplitudes are of those patterns. We'll first consider a very simple data set, which consists of just a single sinusoidal oscillation. In this homework, you'll learn how to read the data set. Based on the graphed data, you should be able to tell what the amplitude and period is. In the second homework, you'll learn how to get the computer to determine the amplitude and period.

[A neat introduction to this is here]

[i]Even though it does this once per day, the result is a twice per day increase in the air pressure because at night the sun's heating is zero, rather than oscillatory. This was actually the subject of my first published paper (search 'atmospheric tides Cohen' and it will probably come up).

D.1.2 Opening the data file

The first data series you'll examine is a simple sinusoidal data set.

- Download the testdata.txt file and make a note of where you put it.
- If using the glowscript interface, create a program with only this one line (other than the header):[ii]
  ```
  finput = read_local_file()
  ```
- Run the program. It should prompt you to select a file that is on your computer. Select the "testdata.txt" file. The program should then end.

D.1.3 Reading the data from the file

Now that we know how to open a file, we now need to read the information that is in the file. The `read_local_file` function extracts information about the file and stores that as a string of characters, which the line above assigns the name `finput`, which is just an arbitrary name (you can use a different name you want).

When I say that `read_local_file` extracts information about the file, I mean it extracts more than just the file contents. It also includes information like the name, size, type and date of the file. Everything is in our `finput` variable. We can then use a particular suffix to get the particular piece of information we want. For example, `finput.date` gives you the date of the file. We want the `text` part of the file, which contains the data we want to analyze. The following line does this and places the text into a variable called data.

```
data = finput.text
```

Add the `finput` line to your code then print out the contents of `data` by adding a `print` command (see previous chapters). You should see that `data` is a single long text that contains all the values separated by commas (like `10.1, 8.3, 5.8,`). There are actually 1000 data points contained in the "testdata.txt" file.

[ii]The way Python handles data files depends upon the Python interface you are using. Glowscript is more complicated than most because glowscript runs within a browser and browsers typically restrict the access of files on your computer as a matter of security. Consequently, you need to instruct the program to bring up a button on the screen that the user can click on to choose the correct data file. Otherwise, you can probably just use a line like `finput = open(‘‘testdata.txt’’,’ ‘‘r’’)`.

D.1.4 Breaking up the data into an array

Before we can plot the data, we need to convert it to an array of individual values. In other words, we need to break up the single long text (that is `data`) into pieces, using the comma as a delimiter. This is done via the `.split` suffix.

```
subdata = data.split(',')
```

The variable `subdata` is an array that contains the values. In other words, `subdata` contains a series of values, with each value referenced by a number within the `subdata` parameter, so `subdata[0]` is the first value in the list, `subdata[1]` is the second value in the list, and so on.

- Add the `.split` line to your code then print out the contents of `subdata` by modifying your print command. You should see that `subdata` consists of a series of numbers.

There are two things to notice about the `subdata` values. First, the values are within a single set of brackets `[...]`, which indicates that the values are part of an array. Second, each value is contained in quotes `' '`, which indicates that each value is being interpreted as text (a set of characters), not a number. Before the program can work with the numbers, we first need to convert the text values to real numbers. This can be done as follows, which creates a new array named `signal` that contains the same values as in `subdata` but as numbers rather than as text.

```
signal = []
n = len(subdata)
for i in range(n):
  signal.append(float(subdata[i]))
```

The first line creates the new `signal` array. By using the brackets we are instructing the program to interpret `signal` as an array (just with no values yet, which is why there are no values within the brackets). The second line determines the number of values within the `subdata` array (and assigns that number to `n`). Lines 3 and 4 go through each value within the `subdata` array, converting it to a real number (via the `float` function[iii]) and then appending it to the `signal` array.

[iii] The word "float" comes from the fact that a real number is a **floating point decimal**.

- Convert the `subdata` array to an array of real numbers via the `float` command then print out the contents of `signal` by modifying your `print` command.

As you've probably seen, it is hard to get a sense of any pattern in the data values from a printout "dump". To make it easier to see, we'll replace the `print` command with a graph. There are three steps to creating a graph, which is accomplished by the following lines.

```
graph1 = gdisplay(title = 'Time series')
f1 = gcurve(color=color.red)
for j in arange(n):
  f1.plot(pos=(j,signal[j]))
```

The first line sets up the graph. There are many different parameters you can set. Here I only set the title. Other parameters include the titles for the axes (via `xtitle` and `ytitle`) and the maximum and minimum values for each axis (via `xmin`, `xmax`, `ymin` and `ymax`). The second line sets up the curve. Again, there are many different parameters you can set. Here I only set the color. Lines 3 and 4 plot each data value in the `signal` array.

- Replace the print command with a graph. You should find that "test-data.txt" data represents a sinusoidal function of amplitude 10 and period 40.

D.1: Copy the graph to a file and upload your file.[iv]

D.2 Examining the data series for embedded functions

From section D.1, you should be able to tell that "testdata.txt" contains 1000 values, representing a sinusoidal function of amplitude 10 and period 40. Our task in this section is to get the computer to come to that same conclusion. You'll add to the program that you wrote for section D.1.

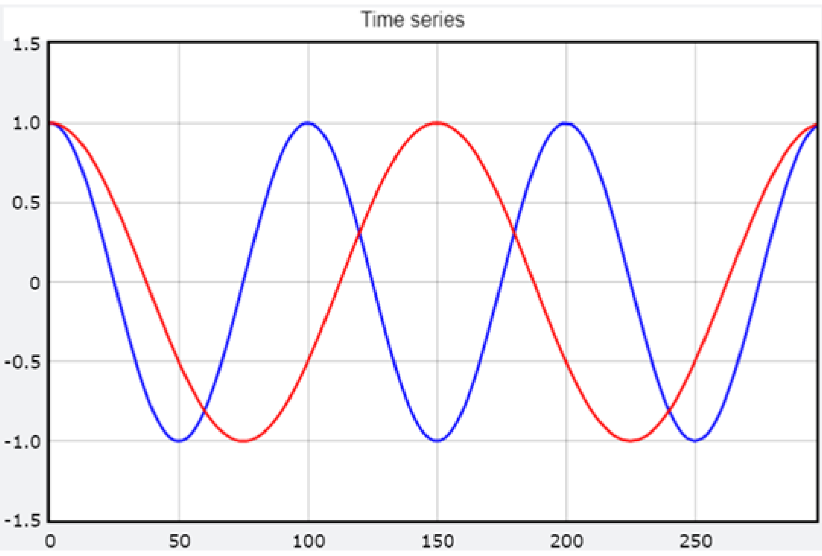
[iv] You can either 'snip' a portion of the screen (the short-cut in Windows is Shift+⊞+S; for Mac, it is Shift+⌘+4) or copy the entire screen (the short-cut in Windows is ⊞+PrtScn or Fn+⊞+Spacebar; for Mac, it is Shift+⌘+3).

D.2.1 Determining if a particular function is present

The basic idea is as follows: if a particular function is present within our `signal` data then there will be a **correlation** between the function and the data. For example, suppose the data series is represented by $f(t)$ and we want to know if the function $g(t)$ is a match. To do this, we go through each data point and multiply the value of $f(t)$ by the value of $g(t)$ at that point. If the two functions aren't correlated at all, the sum of those products should be zero. Otherwise, there is a correlation, suggesting a match.

To understand why a sum equal to zero means the two are not correlated, we first need to recognize that our data set, $f(t)$, and the test function, $g(t)$, have an average value equal to zero (since they are sinusoidal functions).[v] Next, we need to recognize that each function is positive as often as negative. If the two functions are totally unrelated, then, the product of the two (at a particular t) will be positive as often as negative and so the total sum of all the products will be zero. That is why we can use the sum of the products to determine if there is a correlation or not.

To illustrate, consider the following graph of two functions.



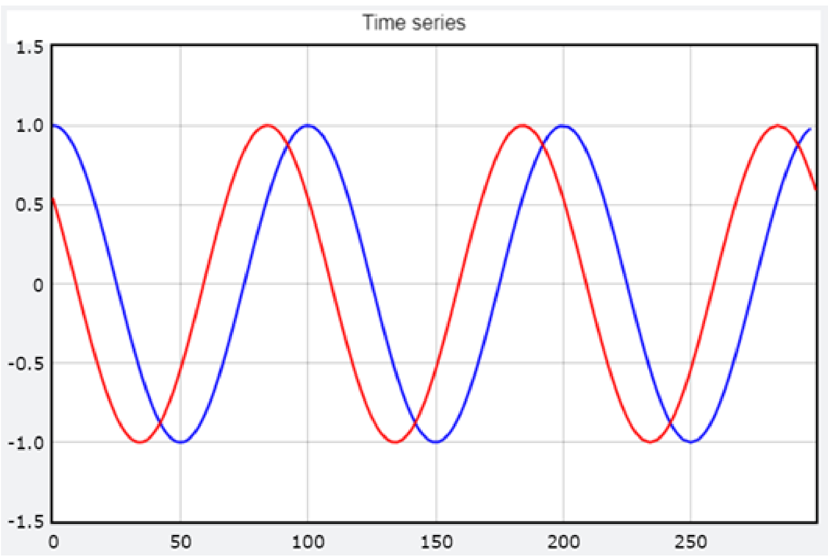
[v]Technically, only the test function must have a zero average, which it has in this case. For example, if the data set average was A then we can represent it as $A + f(t)$, where the average of $f(t)$ is zero. Multiplying by our test function, we have $A \cdot g(t) + f(t) \cdot g(t)$, and since the average $g(t)$ is zero, the average of $A \cdot g(t)$ must also be zero.

The red function has a period of 150, so two cycles of red appear on the graph. The blue function has a period of 100, so three cycles of blue appear on the graph. Both functions start out positive, so their product is initially positive. At around 25, the blue function becomes negative yet the red function is still positive, so their product is now negative. This continues until around 38, when the red function also becomes negative, making the product positive again. If the product is negative as often as positive then the sum of all those products should be zero.[vi]

D.2: As noted above, the product of the two plotted functions is positive between 0 and 25, whereas it is negative between 25 and 38. Identify the portions during which the product of the two functions is positive (you'll have to estimate unless you create your own graphs of these two functions).

D.3: During what fraction of the entire 300 time period is the product of the two functions positive? Why does this suggest that the sum of all the products throughout the entire time period is likely to be zero?

D.4: Below is a graph of two functions that have the same period. Repeat the process described in questions D.2 and D.3 but for the graph below. During what fraction of the entire 300 time period is the product of the two functions positive? Why does this suggest that the sum of all the products throughout the entire time period is unlikely to be zero?



[vi]As mentioned earlier, this is because the mean of $g(t)$ is zero.

Let's utilize this idea to test whether the function $\cos(2\pi t/40)$ is contained in our signal data. We already suspect this will be a match based on the graph from section D.1 since $\cos(2\pi t/40)$ represents a sinusoidal oscillation with a period equal to 40 (since when $t = 40$, the function has a value equal to $\cos 2\pi$, which is the same as $\cos 0$). To see if it really is a match, we need to go through the entire signal data array, multiplying the value at each point by the value of $\cos(2\pi t/40)$ and then determine the sum of those products. This is accomplished by the following set of code.

```
Period = 40.
correl = 0.
for j in range(n):
  correl += signal[j] * cos(2*pi*(j+1)/Period)
print ('correlation:',correl)
```

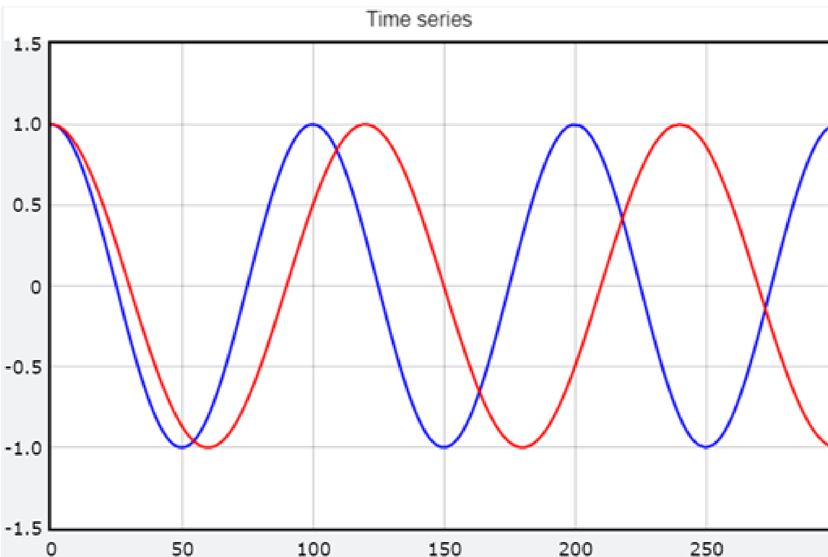
Recall that ''+='' is short-hand for telling Python to add the quantity to its current value. In other words, `A += 1` is equivalent to `A = A + 1`.

D.5: Add the code described above. When you run this, what value of correl is printed out? Does this suggest that $\cos(2\pi t/40)$ is present within our data? Why or why not?

D.6: Repeat the program but edit Period to be something other than 40 that evenly divides into 1000, like 20 or 50. What period did you use and what value of correl is printed out? Does your value suggest that $\cos(2\pi t/T)$, with T being your new period, is present within the data? Why or why not?

D.2.2 Functions with periods that do not go evenly into the array size

The process described above works but only if the embedded function and the testing function have periods that go evenly into the size of the array. For example, consider the situation plotted in the following graph. You'll notice that the blue function has an even number of cycles (3) but the red function does not (2.5). Even though the two have different periods, you can still end up with a non-zero correlation because of the 'extra' part of a cycle is present, which could lead to more positive products than negative products, or visa-versa.



D.7: Repeat the program but edit `Period` to be something that does not evenly divide into 1000, like 30 or 35. What period did you use and what value of `correl` is printed out? Why is the value not zero?

To ensure that the period we try goes evenly into the 1000 points, we'll specify an integer value for the **frequency** (number of cycles within the data) and then divide the 1000 points by the frequency to get the period. A frequency of 25 corresponds to a period of 40, while a frequency of 50 corresponds to a period of 20. For example, instead of setting the period to 40, as before, we will instead set the frequency to 25 (since 1000 divided by 25 is equal to 40).

```
Freq = 25.
Period = n/Freq
```

D.8: Modify the program as mentioned above, with a frequency equal to 25 (and the program calculating the period) rather than specifying a period equal to 40. Make sure you delete the `Period = 40` statement you had before. When you run this, what value of `correl` is printed out?

D.2.3 Cycling through all possible sinusoidal functions

It can get pretty tedious testing one frequency at a time. Via a loop, we can test all sinusoidal functions that have an integer number of cycles within the

domain. To do this, delete the two lines listed above and instead embed the `j` loop within another loop, as follows (new lines in bold; others may have some edits):

```
for i in range(n):
  Period = float(n)/float(i+1)
  correl = 0.
  for j in range(n):
    correl += signal[j] * cos(2*pi*(j+1)/Period)
  print ('Freq:  ',i+1,'; correlation:  ',correl)
```

When you run the updated code, you'll get a printout of 1000 `correl` values, one for every frequency from `1` to `n`. A graph of the results would be much easier to interpret. In this case, we'll set up a bar graph, which is accomplished in the same way as the 3-step process for graphing outlined in section D.1 but with `gvbars` instead of `gcurve` (you can call the plot `f2`, to distinguish it from `f1`, which you used before). A bar graph is more appropriate since the correlation is discontinuous with frequency (meaning that we only expect non-zero correlation values for the specific frequencies that are embedded in the data).

The first two lines, which set up the graph and the bars, should be added before the `i` loop, since they only need to be set up once. Title the graph something like ''Frequency distribution" rather than ''Time series"). Whereas before you needed a loop to plot each point, here you don't, since you already have a loop that goes through each frequency (the i loop). Consequently, you just need to replace the `print` line above with a `plot` statement that looks like the following.

```
  f2.plot(pos=(i,correl))
```

D.9: Set up your code to create a bar graph of the `correl` for each frequency from `1` to `1000`. According to your graph, how many frequencies have a non-zero `correl` value?

D.2.4 Setting the frequency domain to $n/2$

When you run the code, you should see that the graph is symmetric around a frequency that is half of n (note that a frequency of $n/2$ has a period equal to 2). Indeed, there is no need to test frequencies greater than $n/2$. Consequently, you can change the `for i in range(n)` to `for i in range(n/2)`.

D.10: 10. Run the code and verify that only one frequency has a significant correlation with the data series. Copy the graph to your homework document or upload it as a separate file.

D.3 Introduction to Fourier Analysis

D.3.1 A problem with the phase

The sinusoidal function that I used to create the "testdata.txt" file had an amplitude of 10 and a frequency of 25. Given that, you might wonder why we get a negative correlation when we multiply the series by a cosine function with the same frequency of 25.

The answer has to do with the **phase** of the sinusoidal function. Our "test" cosine function has a phase of zero, meaning that it has a value of $+1$ when the angle is zero. If we make the phase $\pi/2$ then it would start with a value equal to zero (like the sine function). If we make the phase π then it would start with a value equal to -1.

The function embedded in the test data has a phase close to π. This means that there is an out-of-phase alignment between our test function and the embedded function (i.e., the two have opposite values most of the time). That leads to a negative correlation.

- To see how the phase impacts the correlation, calculate the correlation between the "testdata.txt" data and $\sin(2\pi t/40)$, in addition to the correlation between the "testdata.txt" data and $\cos(2\pi t/40)$. The two are identical except that one is shifted by $\pi/2$. You should find that both correlations are negative but the correlation with the sine is smaller, since the phase is closer to $-\cos(2\pi t/40)$ than it is to $-\sin(2\pi t/40)$.

D.3.2 Accounting for the phase

Because of the issue with the phase, it is best to calculate BOTH the correlation with sine and the correlation with cosine. Regardless of the phase, at least one will show a correlation if the same frequency is in the data. The phase can be determined by the relative strength of each.

Rather than plot both, however, one can plot a single COMBINED value. The combined value is obtained by squaring both the sine and cosine correlations and taking the square-root of the sum.[vii] At the end, when you have the sum of all those combine values, divide by the total number of values, `float(n)`. This will then give an average correlation per data point.

- Change your plot to get the new correlation value (based on both the sine and cosine correlations, as described above). You should get a correlation equal to 5 at a frequency of 25, and zero for all other frequencies.

D.3.3 Determining the amplitude

When you run the program with the "combined" correlation, you may have noticed that the plotted value is half the amplitude of the signal I used to create the data set (10). That is not a coincidence. The average value of the $\cos^2\theta$ is one-half (as is the average value of $\sin^2\theta$) when measured over an integer number of periods. Consequently, the method used above gives a value equal to half the amplitude of the function.

- Change your plot so that you multiply the correlation by two. Rerun the program. You should get a correlation equal to 10 at a frequency of 25 (and zero everywhere else), matching the amplitude of the signal in the data set.

D.3.4 Analyzing data that is "hidden"

To show how robust this process is, download the files testdata_random1.txt, testdata_random2.txt and workfile.txt.

D.11: Run your code on the "testdata_random1.txt" file data, which consists of the same sinusoidal function as "testdata.txt" (amplitude 10 and frequency 25) but has, in addition, some noise, made up of a random value between -20 and $+20$ at each point. Your program should create two graphs, a time series constructed from the "testdata_random1.txt" data and a bar graph showing the frequency distribution within the data. Copy both graphs to your homework document (or upload as a separate file).

[vii] This is similar to the Pythagorean Theorem: $A^2 + B^2 = C^2$.

(a) Can you clearly identify by eye the embedded sinusoidal signal (amplitude 10 and frequency 25) in the time series graph?
(b) Does your program produce a frequency distribution graph that identifies the correct amplitude (10) and frequency (25) of the embedded signal?[viii]

D.12: Run your code on the testdata_random2.txt file data, which is like testdata_random1.txt except that the noise consists of random values from −50 to +50 (instead of −25 to +25). Copy the two graphs it produces (the time series one and the frequency distribution one) to your homework document (or upload as a separate file).
(a) Can you clearly identify by eye the embedded sinusoidal signal (amplitude 10 and frequency 25) in the time series graph?
(b) Does your program produce a frequency distribution graph that identifies the correct amplitude (10) and frequency (25) of the embedded signal?[ix]

D.13: The real power of this analysis is not when we have a single function to analyze but rather when multiple frequencies and amplitudes present along with a random signal, which is what I did with "workfile.txt".
(a) How many different frequencies did I use to create the "workfile.txt" file?
(b) Identify the frequencies of the three largest signals (in terms of amplitude).

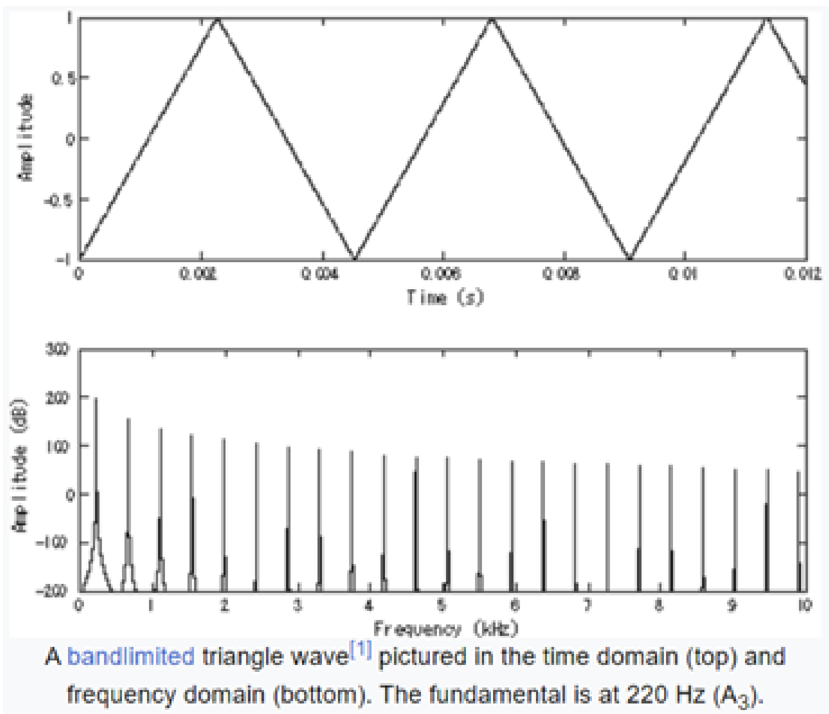
Problems

Problem D.1: Why does the program go through all the *frequencies* from 1 to 500 rather than going through all the *periods* from 1 to 500?

Problem D.2: Consider this Triangle Wave:

[viii]The presence of the random values should introduce small 'noise' throughout the frequency spectrum.

[ix]If the noise is too great, the estimate of the amplitude can be off.



A bandlimited triangle wave[1] pictured in the time domain (top) and frequency domain (bottom). The fundamental is at 220 Hz (A_3).

(a) Describe what it is about the time domain plot (top) that predicts the first peak at 0.22 kHz in the frequency domain plot (bottom).
(b) Given your answer in (a), why are there other peaks in the frequency domain plot (bottom)?

Problem D.3: The testdata.txt file was constructed using a sinusoidal function equal to $10\cos[2\pi(t/40+0.4)]$. Suppose it was instead constructed with $10+10\cos[2\pi(t/40+0.4)]$. How would that impact the frequency bar plot created by our program (without any further edits to the program)?
a. It won't impact it at all; the bar plot would look exactly the same with a single bar of height 10
b. The plot would look the same except that the single bar would have a height equal to 20 instead of 10
c. The entire plot would be covered by bars of height equal to 10
d. The entire plot would be covered by bars of height equal to 10 except for one part of height 20
e. None of the above

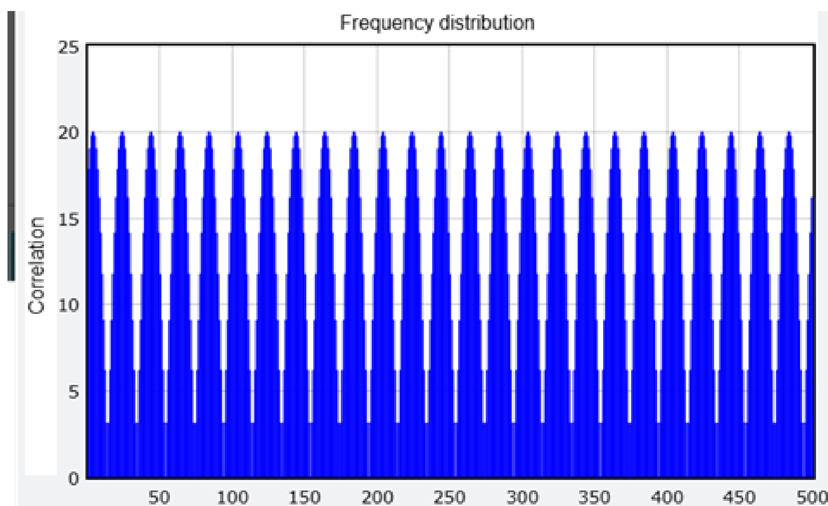
Problem D.4: The testdata.txt file was constructed using 1000 points of a sinusoidal function equal to $10\cos[2\pi(t/40+0.4)]$, where t are the integers

from 1 to 1000. In our code, we multiplied the data by a cosine function and a sine function, with `Acos` being the sum of the products with the cosine.
(a) Suppose the testdata.txt file was instead constructed with $10\cos[2\pi(t/40)]$. Estimate the new value `Acos` and provide a rationale for your estimate.
(b) Suppose the testdata.txt file was instead constructed with $10\cos[2\pi(t/40)+\pi/2]$. Estimate the new value `Acos` and provide a rationale for your estimate.

Problem D.5: The following code is a student's attempt to create the Fourier analysis program.

```
for j in range(n/2):
  period = float(n)/float(j+1)
  Acos = 0.
  Asin = 0.
  for i in range(n):
    Acos += 2.*signal[j]*cos(2*pi*float(j+1)/period)/float(n)
    Asin += 2.*signal[j]*cos(pi/2 +
2*pi*float(j+1)/period)/float(n)
  f2.plot(j+1, sqrt(Acos**2 + Asin**2))
```

When the code is run with our testdata.txt file, the following graph is produced.



Identify the errors in the code and explain how you'd fix the errors.

E. Integration

- Physics context: Rocket launch
- Programming skills:
- Computational skills: Integration methods (1st order, 2nd order, 3rd order, trapezoidal)
- Mathematics skills: Integration

E.1 Context and simple integration scheme

E.1.1 Numerical vs. analytical integration

When you took your calculus class, you learned how to apply principles of calculus to determine an integral. This basically involves two ideas. One idea is that the integral represents an infinite sum of infinitesimals,

$$\int dx = \Delta x$$

and the other idea is some property of differentiation, like the power rule,

$$d(x^n) = nx^{n-1}dx$$

The process involves first obtaining an equation that represents the integral and then plugging in the limits to find the value of the integration for the case we are interested in. For example, if the function was $y = x^2$, you can apply the two principles above to get the following:

$$\int y dx = \int x^2 dx = \int d\left(\frac{x^3}{3}\right) = \Delta\left(\frac{x^3}{3}\right)$$

To find the value of this integral, we then plug in the limits of x.

We can instead use the computer to find the value of the integral. As with root finding and the other examples we've examined computationally, the

computer won't first derive an equation that gives the answer. When we use the computer to do it, we call that **numerical integration**, compared to **analytical integration** when we use principles of calculus to first derive an expression for the integral and then plug in the values.

E.1.2 Free fall

To illustrate how we integrate analytically vs. numerically, I'll start with a case of free fall. You should already know how to do this analytically, which allows us to more easily compare the analytical and numerical approaches.

During free fall, the only force acting on an object is that due to gravity. As long as the object is not moving fast enough that drag becomes significant and as long as the distance fallen is not great compared to the distance to the center of Earth, we can treat the net force and thus the acceleration as being constant, which we'll indicate as g.

From the definition of acceleration, $a = dv/dt$. Since $a = g$, we can write that $dv = gdt$. Integrating both sides, we can obtain an expression for the change in velocity during a time period from 0 to T:[i]

$$\int_0^T dv = g\int_0^T dt$$

$$\Delta v = gT \tag{E.1}$$

E.1: Use equation (E.1) to determine the speed of an object, starting in rest, after free fall for 20 seconds. Assuming a free fall acceleration equal to 9.8 m/s^2.

Now let's look at how to do the integration *numerically*. Keep in mind that we typically turn to the computer when we *don't* know the analytical solution (after all, if we did know, we wouldn't need the computer). However, we will consider the free fall case (which we are able to solve analytically) so we can see if our numerical solution is correct or not. Our technique should be able to work for any situation, whether solvable analytically or not, but if we didn't have an analytical solution when testing out our code, we wouldn't know if our numerical technique was correct.

[i]Note that g is constant so I can take it out of the integral in the same way we can bring the "3" out of the expression "$3x + 3y$" to get "$3(x + y)$".

When doing numerical integration, it is important to recognize that the computer is not supposed to figure out the analytical solution. Instead, the task is to just apply the idea that the integral is the *sum of all the tiny pieces*. The *numerical* way, then, is to let the computer do the summation for us. Such a process would be too tedious to do ourselves, which is why we use principles of calculus instead, but that doesn't mean the computer can't do it.

Recall that $dv = adt$ (from the definition of acceleration) so we have:

$$\Delta v = \int_0^T a \; dt \tag{E.2}$$

When we integrate computationally, we let the computer break up the total time period $\Delta t = T$ into lots of little time periods called **time steps**, each of length δt, and then have the computer calculate $a\delta T$ for each time step and add them all up to get Δv.

To start us off, we'll use the Python program E:SimpleIntegration.

When you run the Python code, you should get -196 m/s (negative because the program uses upward as positive), which corresponds to the velocity of an object, released from rest, in free fall for 20 seconds.

E.2: As mentioned above, the numerical method is to break up the period into lots of little time periods called time steps. In the code, what is the time step value used for this integration?

E.3: How does the code know how many pieces to add together? Describe what the code is doing mathematically. Don't just point to the line number.

E.1.3 Rocket lift-off with constant thrust

Now that we've done a very simple case, let's make it slightly more complicated by considering the motion of a rocket being launched. A rocket lifts off because its thrust is greater than the gravitational force on it. Let's assume, for the time being, that the thrust is constant, with a value per mass equal to f (I'm using lower-case to represent the force per mass, rather than a capital F for the force itself). Since the gravitational force per mass is g, the

acceleration of the rocket is (using positive as upwards):

$$a = \frac{F_net}{m} = \frac{F_{\text{thrust}} - mg}{m} = \frac{F_{\text{thrust}}}{m} - \frac{mg}{m} = f - g$$

E.4: Use the definition of acceleration and methods of calculus to obtain the analytical expression for the velocity v of a rocket that lifts off from rest with a constant thrust per mass equal to f. This should be relatively easy, since the thrust is constant, just like the gravitational force. Then use your analytical solution to determine the rocket's velocity 20 seconds after liftoff, assuming a thrust per mass f equal to 300 N/kg.

Copy the E:SimpleIntegration code to your own account so that you can edit it. Modify the code so that it includes a thrust equal to 300. Before moving on, make sure that when you run the code, you get the same answer as the analytical solution.

E.1.4 Rocket lift-off with constant thrust and decreasing fuel mass

So far, things have been relatively simple because the object's acceleration is constant. We only obtained a constant acceleration because of the following assumptions:

A. We assumed the gravitational force is constant, yet the gravitational force decreases as the rocket moves away from Earth.
B. We assumed no air resistance, which depends on the speed of the rocket and the air density (and both vary with time).
C. We assumed an unchanging rocket mass, yet the mass decreases as it uses up the propellant.

Given how weak these assumptions are for a real rocket, our analysis presented so far is not very realistic. Let's make it slightly more realistic by addressing assumption C (we'll keep assumptions A and B just to make it still solvable analytically).[ii]

To address assumption C, we'll consider a rocket that is made up of two parts: a structural part of mass M_0 that remains constant and a fuel part of mass M_{fuel} that decreases with time as the fuel is used up.

[ii]For an in-depth analysis of rocket equations, see this page.

We'll still assume constant thrust, but the thrust per mass will change, since the mass will change. To address this, we set the thrust per structural mass to be constant and equal to f. Thus, the thrust itself is fM_0 and the gravitational force is $g(M_0 + M_{\text{fuel}})$. From Newton's second law, that means the acceleration is:

$$a = \frac{F_{\text{net}}}{m_{\text{total}}} = \frac{fM_0 - g(M_0 + M_{\text{fuel}})}{M_0 + M_{\text{fuel}}}$$

Using α (alpha) for the fuel to structure mass ratio (M_{fuel}/M_0), this can be rewritten as follows:

$$a = \frac{f}{1+\alpha} - g$$

Suppose that T_{th} is the time it takes for the rocket to use up its fuel, then M_{fuel} decreases linearly with time as $M_{\text{fuel}}(t) = M_{\text{fuel}}(0)[1 - t/T_{\text{th}}]$ until $t = T_{\text{th}}$, after which time $M_{\text{fuel}} = 0$. Using α_0 is the initial fuel to structure mass ratio, we have (for $t \leq T_{\text{th}}$):

$$a = \frac{f}{1 + \alpha_0(1 - t/T_{\text{th}})} - g \tag{E.3}$$

E.5: Based on equation (E.3), is the acceleration of the rocket increasing with time or decreasing with time? Explain how this follows mathematically from equation (E.3) and also explain how the result is consistent with Newton's second law.

Equation (E.3) is a little more complicated than what we had with constant thrust but it can still be solved analytically (applying principles of integration) to get the following analytical solution:

$$v = v_0 - gt - \frac{fT_{\text{th}}}{\alpha_0} \ln \frac{1 + \alpha_0 - \alpha_0 t/T_{\text{th}}}{1 + \alpha_0} \tag{E.4}$$

E.6: Derive equation (E.4) from equation (E.3). To do this, you'll need to recognize that the integral has the form:

$$\int \frac{a}{b - cx} dx$$

Substituting $u = b - cx$ leads to the following solution:[iii]

$$-\frac{a}{c}\Delta \ln(b - cx)$$

E.7: Use equation (E.4) to determine the speed of the rocket 20 seconds after liftoff, with thrust per permanent mass f equal to 300 N/kg, an initial fuel mass that is nine times more than the permanent mass (i.e., $\alpha_0 = 9$), and time to shutoff T_{th} equal to 20 seconds.

We'll now modify the code to do the integration numerically. This is best done by first defining some parameters and then using those parameters in your calculation. For example:

```
g = 9.8 # m/s2
alpha = 9.
Tth = 20.  # seconds
f = 300.  # N/kg
```

You can then modify the acceleration function according to equation (3):[iv]

```
def accel(time):
  if time>Tth:
    return - g
  else:
    return f / ( 1.  + alpha * (1.  - time / Tth ) ) - g
```

You will also need to modify the call to the acceleration function, since previously the acceleration didn't depend upon time so it was called as `accel()`.

[iii]This can be seen by recognizing that $du = -cdx$. Replacing $b-cx$ with u, and replacing dx by $-du/c$, we get that the integrand is $(-a/cu)$ and since a and c are constants, we can take those out of the integral and we get and integral that is easy to solve:

$$-\frac{a}{c}\int \frac{du}{u} = -\frac{a}{c}\Delta \ln u$$

Substituting back in $u = b - cx$, we get the expression $-(a/c)\Delta \ln(b - cx)$.

[iv]Notice that the definition includes an `if` statement that checks to see if the fuel has been used up. In our case, we'll only predict the speed at the moment the fuel shuts off but the `if` statement is there in case the user wants to predict beyond that (i.e., if the time is greater than `Tth` then only the gravitational force is acting and the rocket is in free fall). Due to rounding in the 15th digit, it is probably best to compare the time with the something like `Tth*(1 + 1e-15)`.

Now it depends upon time so you need to call it as accel(i*dt). Notice that this determines the time by multiplying dt by i. One could instead create a variable and add dt to it each time but, due to the internal rounding in the computer, that just increases the rounding error over time. Multiplying by i (which is an integer and therefore not rounded) keeps the rounding in the 15th digit. The rounding problem is also the reason why we are doing a loop for nint steps rather than keeping track of the time by adding dt to it each time and stopping when the time gets to the final time.

Also add some code to calculate the analytical solution as a comparison. It should look something like the following.

```
##### Analytical solution
print ("**Analytical solution**")
vf = vi-g*tf \
  -(f*Tth/alpha)*log((1+alpha-alpha*tf/Tth)/(1+alpha))
print ("final velocity =",vf,"m/s')
```

E.8: When you run your code, what values do you get for the analytical solution and the numerical solution? Do they agree?

E.1.5 Impact of step size

The reason the numerical and analytical solutions are different in this case is because the acceleration is not constant. Remember that integration consists of an infinite sum of infinitesimal values. In our numerical scheme, we used small time steps (1 s) but those are not infinitesimal time steps.

E.9: Redo the numerical solution but 10^{-2} s and then with 10^{-4} s. What are the calculated speeds for those time steps? Which is closest to the analytical solution: the one with a 1 s time step, the one with the 10^{-2} s time step, or the one with a 10^{-4} s time step?

You should find that using smaller time steps results in a solution that is closer to the analytical solution. To understand why, consider that the acceleration we actually need is the *average* acceleration during the time step. In this code, the acceleration is based on the mass of the rocket at the *beginning* of the time step, not the *average* acceleration *during* the time step. The smaller the time step the closer the calculated acceleration (at the beginning of the time step) is to the average acceleration during the time step.

E.10: While smaller time steps get you closer to the analytical answer, the numerical answer is always smaller than the analytical answer. Is this (always being smaller) consistent with your answer to Question E.5 (acceleration increasing with time or decreasing with time)? Why or why not?

At this point you might wonder why we don't just use a really, really small time step. You can try it with even smaller values but you should find that it takes longer to carry out the calculations when a smaller time step is used because it requires a larger number of steps overall to get to the final time (a factor of ten more iterations for each factor of ten decrease in time step).

E.11: Why wouldn't we want to use a really small time step, even though doing so will give us a better solution?

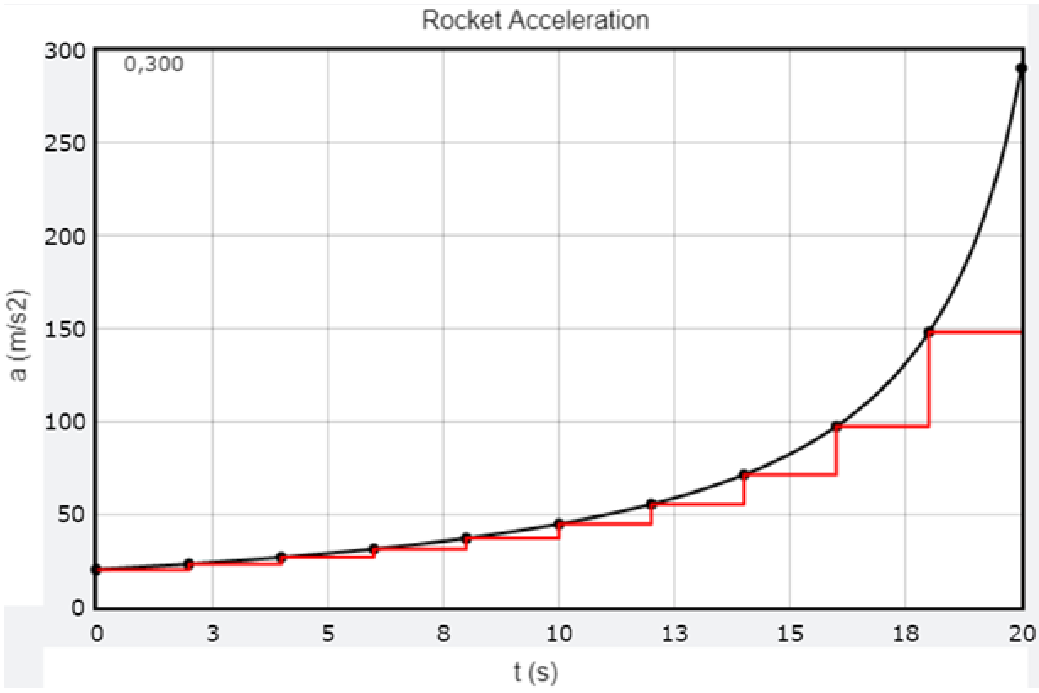
E.12: Why does a smaller time step get us a *better* answer here but it resulted in *worse* answers in section A.3 when we examined the impact of resolution on a long summation (something similar to what we are doing here)?

E.2 Numerical methods of integration

E.2.1 Initial value method

As mentioned in section E.1, decreasing the size of the time step gets us closer to the correct value but the numerical calculation is always a little less than the analytical value. This is because we used a numerical method that utilized the acceleration valid at the *beginning* of each time step to predict the velocity at the end of each time step. To get the right value, the numerical method should use the *average* acceleration during each time step.

To illustrate this, consider the plot below, where the black curve is the acceleration of the rocket during the twenty seconds. As you can see, the acceleration increases over time because the rocket's mass decreases (as it uses up the fuel).



In addition to the black curve, I added a red step-like curve. This represents the integration that would occur if we used 2-s steps and the initial acceleration method of section E.1. When we multiply the acceleration by the time step, we get what would be analogous to an area on the graph, where the acceleration is the height and the time step is the width of a rectangle. Adding up all the rectangles, we get an area, but that area is *less* than the area under the black curve.

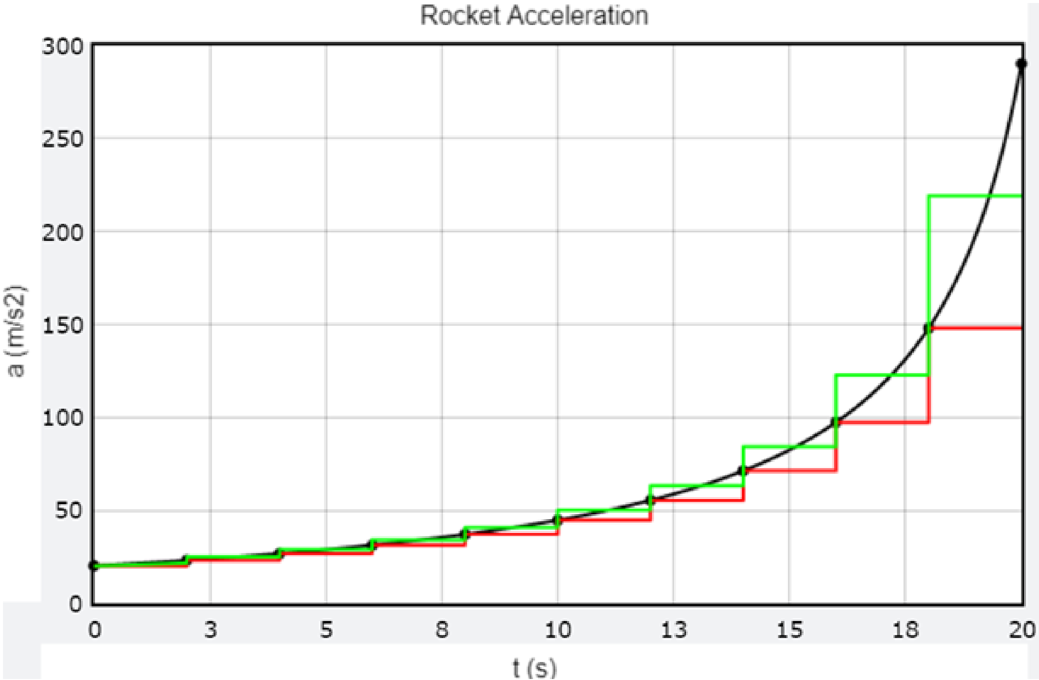
By using the acceleration that is valid at the beginning of each time step, we end up underestimating the average acceleration during the time step, resulting in a numerical value that is *less* than the theoretical value. As you might expect, using the acceleration that is valid at the *end* of each time step gives an error in the other direction (a numerical value that is always a little *more* than the theoretical value).

E.2.2 Midrange (trapezoidal) method

Given that we end up *underestimating* the average acceleration if we use the value valid at the beginning of each time step, and we end up *overestimating*

the average acceleration if we use the value valid at the end of each time step, one obvious way to solve this issue, perhaps, is to use the *average* of the beginning and ending values. I call this the **midrange** method because the acceleration value in this case is the midrange between the beginning and ending accelerations during the time step.

This is illustrated in the graph below, where the green step-like curve has been added. The green step-like curve is like the red step-like curve, in that it reflects what happens with 2-s time steps. However, whereas the top of the red curve corresponds to the acceleration value at the *beginning* of each time step, the top of the green curve corresponds to the *midrange* acceleration value during each time step.



You should be able to see that the area under the green curve provides a better match to the area under the black curve than the area under the red curve does.

Keep in mind that the midrange is not necessarily equal to the *average during the interval.* To see why, consider an acceleration that starts at 0 at time zero, increases linearly to 1 m/s^2 at 1 s, 2 m/s^2 at 2 s and then remains at

2 m/s^2 at 3 s and at 4 s. The midrange during the four seconds is 1 m/s^2, since it started at zero and ended at 2 m/s^2. The average value, though, is actually 1.5 m/s^2 (see footnote[v] for why). Certainly, the midrange of 1 m/s^2 is a better estimate to the average acceleration (1.5 m/s^2) than the initial value of zero. However, it is *not* the average.

In fact, the midrange will only *necessarily* equal the average if the function is linear (i.e., straight line on graph). For our situation, the function is not linear, as illustrated by the curved black line. Indeed, the curve is such that, during any period, it spends more time below the midrange value than above it. For this reason, in our case the midrange probably *over-estimates* the average acceleration during each time period.

Note: This is usually called the **trapezoidal method** (or **composite trapezoidal method**) because when graphed the integral appears as an area under trapezoidal approximations to the curve.

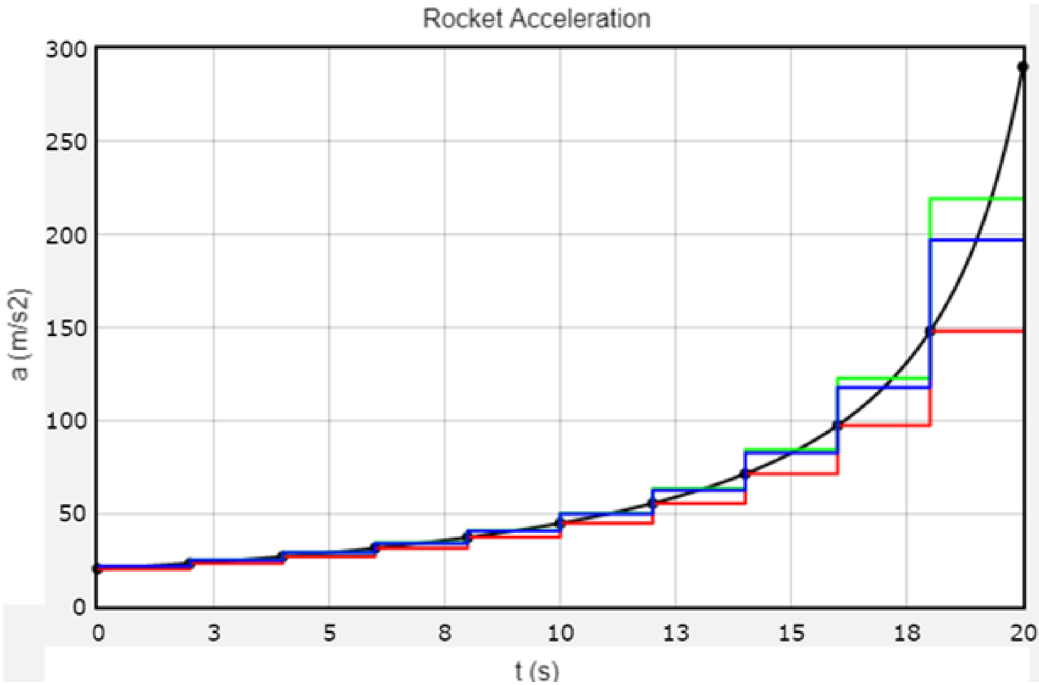
E.13: Given the discussion above, do you expect the midrange method to give a velocity value that is *lower* than the analytical solution, *higher* than it or *equal* to it? Provide your reasoning.

E.2.3 Midpoint method

A second way to fix the integration error is to use the acceleration value that is valid at the middle of the time interval. This is called the **midpoint method** (or **composite midpoint method**).

This is illustrated in the graph below, where the blue step-like curve has been added, with the top of the blue curve corresponding to the midpoint acceleration value during each 2-s time step.

[v] To see why, consider that the average value during the first 2 seconds is 1 m/s^2, when the acceleration is linearly increasing from zero to 2 m/s^2. In comparison, the average value during the time from 2 s to 4 s is 2 m/s^2, when the acceleration is constant at 2 m/s^2. The average of those two is 1.5 m/s^2.



As with the midrange value, the midpoint value need not be equal to the average acceleration. For the example given earlier, the midpoint value would be 2 m/s^2 because that is the acceleration right at 2 s (halfway through the 4-s time period). Like the midrange value in the example, this is not the average value, which is actually 1.5 m/s^2.

E.14: Given the discussion above, which method do you expect will give a velocity value closer to the analytical solution: the midrange method or the midpoint method? Provide your reasoning.

Copy the integration section of your code twice, once for each additional method (midrange and midpoint). Title each one appropriately and remember to reset `vi` for each. For the midrange method, you'll need to replace the `accel(i*dt)` statement with one that uses the average of the initial and final accelerations during the time step, like `0.5*(accel(i*dt)+accel((i+1)*dt))`. For the midpoint method, you'll need to replace the `accel(i*dt)` statement with `accel((i+0.5)*dt)`.

E.15: Run the code with the four calculation methods (the one analytical method and the three numerical methods – using the acceleration at the beginning of time step, midrange and midpoint) for a time step of 1 s. What are

your results? Do the results agree with your predictions made in Questions E.13 and E.14? If not, explain what was wrong with your reasoning.

E.16: Rerun the code, first with a time step of 1 s then progressively smaller time steps of 10^{-2} s, 10^{-4} s, and 10^{-6} s. Does a smaller time step improve the integration for the velocity? Why would this be?

E.2.4 Taylor series expansions

It turns out that the initial value method (section E.1) is a consequence of the first-order Taylor series expansion and the midpoint method is a consequence of the second-order Taylor series expansion. We saw in chapter A that every function can be written as an expansion of terms called the Taylor series, where each term involves the derivative of the function:[vi]

$$v(t+\delta t) = v(t) + \dot{v}(t)\delta t + \frac{1}{2}\ddot{v}(t)(\delta t)^2 + \frac{1}{3!}\dddot{v}(t)(\delta t)^3 + \frac{1}{4!}\overset{4}{\dot{v}}(t)(\delta t)^4 + \cdots$$

Whereas before I indicated the function as $f(x)$, here I've indicated the function as $v(t)$ to represent the velocity of a rocket.

Remember from before that if you use the *entire* series then the right side will be exactly *equal* to the left side. In other words, you can use the terms on the right with values at t and the sum will give the value of the function at $t + \delta t$ (left side). If you only use *some* of the terms on the right side then you'll only get a value that is *approximately* what you'd get on the value of the function at $t + \delta t$. Some terms are larger than others, so how close you get will depend on which terms you drop. The higher-order derivative terms are likely smaller because those terms are divided by the factorial of the term number, so dropping them likely has a smaller impact than dropping the lower-order terms.

The **first-order** Taylor Series is simply the series up to the first derivative.

$$\boldsymbol{v(t+\delta t) = v(t) + \dot{v}(t)\delta t} \tag{E.5}$$

[vi]This uses the **dot notation** (introduced by Newton, whereas the prime notation was introduced by Euler) to represent a derivative with respect to time, so $v = \dot{x} = dx/dt$ and $a = \dot{v} = \ddot{x} = d^2x/dt^2$. For derivatives beyond the third derivative, I'm using a number with a single dot instead of adding dots, like $\ddddot{x}$.

As you can see, the first-order approximation is equivalent to the **initial value method**. This is not just an interesting feature of the initial value method. By recognizing that it is equivalent to the first-order approximation, we can estimate the error, as the error in $v(t + \delta t)$ is equal to the sum of the remaining terms that have been dropped. Assuming higher order terms are smaller due to the n factorial term in the denominator and the higher power of δt (with δt being less than one), we expect the error to be approximately the value of the first term that has been dropped. Consequently, the error for the first-order approximation would be $\ddot{v}(t)(\delta t)^2/2$.

Assuming the second derivative is on order equal to one, the error is approximately $(\delta t)^2/2$ for each time step. Multiply by $n = \Delta t/\delta t$ to get the approximate total error associated with the integration:

$$\frac{1}{2}(\Delta t)(\delta t) \qquad \text{[approximate first order error]}$$

Let's now see what we get when we include the series up to and including the *second* derivative, which is known as the **second-order** Taylor Series:

$$v(t + \delta t) = v(t) + \dot{v}(t)\delta t + \frac{1}{2}\ddot{v}(t)(\delta t)^2$$

It turns out that the second-order Taylor series expansion is equivalent to the *midpoint value* method. To see why, follow the method outlined in chapter A, where we write the second-order expression twice, once for $t + \delta t/2$ and once for $t - \delta t/2$:

$$v(t + \delta t/2) = v(t) + \frac{1}{2}\dot{v}(t)\delta t + \frac{1}{8}\ddot{v}(t)(\delta t)^2$$
$$v(t - \delta t/2) = v(t) - \frac{1}{2}\dot{v}(t)\delta t + \frac{1}{8}\ddot{v}(t)(\delta t)^2$$

Since the second expression uses negative $\delta t/2$ instead of positive $\delta t/2$, the second term has opposite signs but the third terms are both positive, since $\delta t/2$ is squared in that term. We now subtract the second expression from the first to get rid of the first and third terms in the expansion:

$$v(t + \delta t/2) - v(t - \delta t/2) = \dot{v}(t)\delta t$$

Rearranging and shifting t to $t + \delta t$ or, equivalently, adding $\delta t/2$ to the time, we get:

$$\boldsymbol{v(t + \delta t) = v(t) + \dot{v}(t + \delta t/2)\delta t} \tag{E.6}$$

This is the **midpoint value method**, where the value at the end of the period is equal to the value at the beginning of the period plus the product of the derivative at the midpoint and the total period length.

To obtain an estimate of the error associated with the second-order (midpoint value) method, we can repeat the derivation including the third-order term then, at the end, see what needs to be ignored to get the final expression. If you do this[vii], you get that the error is approximately $\dddot{v}(\delta t)^3/24$.

Assuming the third derivative is on order equal to one, the error is approximately $(\delta t)^3/24$ for each time step. Multiply by $n = \Delta t/\delta t$ to get the approximate total error associated with the integration:

$$\frac{1}{24}(\Delta t)(\delta t)^2 \qquad \text{[approximate second order error]}$$

This shows that the second-order (midpoint) method is better than the first-order (initial value) method, not only because of the larger denominator (24 vs. 2) but also because the second-order error is proportional to the *square* of the time step. This also means that cutting the time step impacts the second-order approach more than it impacts the first-order approach (a factor of 100 instead of a factor of 10 when we cut the time step to one-tenth, for example).

E.17: Does the value of the second derivative of v impact the accuracy of the initial value method (equation E.5)? In other words, would a larger second derivative make the method less accurate? Why or why not?

E.18: Does the value of the second derivative of v impact the accuracy of the midpoint method (equation E.6)? Why or why not?

[vii] Including the third order term, and recognizing that $1/3! = 1/6$, $(1/2)^3 = (1/8)$ and $(1/6)(1/8) = (1/48)$, we have:

$$v(t + \delta t/2) = v(t) + \frac{1}{2}\dot{v}(t)\delta t + \frac{1}{8}\ddot{v}(t)(\delta t)^2 + \frac{1}{48}\dddot{v}(t)(\delta t)^3$$

$$v(t - \delta t/2) = v(t) - \frac{1}{2}\dot{v}(t)\delta t + \frac{1}{8}\ddot{v}(t)(\delta t)^2 - \frac{1}{48}\dddot{v}(t)(\delta t)^3$$

Subtracting, and adding $\delta t/2$ to the time, we get:

$$v(t + \delta t) - v(t) = \dot{v}(t + \delta t/2)\delta t + \frac{1}{24}\dddot{v}(t + \delta t/2)(\delta t)^3$$

E.2.5 Integration with higher-order Taylor series

At this point, you may be wondering about two things. First, if we know how to calculate the first derivative, why not calculate the higher-order derivatives as well, thus avoiding the need to ignore the higher-order terms in the first place? Second, if we get a better estimate using the second-order expansion why not use a higher-order expansion like the third-order or fourth-order expansion?

Let's first address the first question. Basically, in our rocket problem, the expression was relatively simple. Indeed, it was simple enough that we could determine the integral analytically and, if we wanted to, we could determine the higher-order derivatives analytically. That would allow us to do the integration with a higher-order expression, which would certainly improve the integration.

However, a more realistic situation – where gravity is not constant and drag is present (which depends on the density, which is not constant) – would have an acceleration whose derivative could not be determined analytically since gravity and density depend on height, which depends on our calculation of velocity. Yet, we can still integrate numerically (with the error discussed earlier).

[We run into a similar problem when modeling the atmosphere as there are many factors that are interrelated, making it impossible to integrate analytically. However, even simple expressions can be impossible or difficult to solve analytically, like $\sin(x^2)$. For such expressions we must use numerical integration.[viii]]

As for the second question regarding why we don't use a higher-order expansion, the problem is that it gets more and more complicated to create expressions that are accurate to higher order yet only use the first derivative. For example, here is the derivation for how we'd do it to *third* order (the final expression is indicated as equation E.8, in bold below).

We start by taking the third-order expression and writing it twice, once for $t + \delta t/2$ and once for $t - \delta t/2$, as we did with how we got the midpoint

[viii] Interestingly enough, whereas $\sin(x^2)$ is difficult, $x \sin(x^2)$ can be integrated analytically pretty easily by substitution (using $u = x^2$).

expression, but to third-order:

$$v\left(t+\frac{\delta t}{2}\right) = v(t) + \dot{v}(t)\left(\frac{\delta t}{2}\right) + \frac{1}{2}\ddot{v}(t)\left(\frac{\delta t}{2}\right)^2 + \frac{1}{6}\dddot{v}(t)\left(\frac{\delta t}{2}\right)^3$$

$$v\left(t-\frac{\delta t}{2}\right) = v(t) - \dot{v}(t)\left(\frac{\delta t}{2}\right) + \frac{1}{2}\ddot{v}(t)\left(\frac{\delta t}{2}\right)^2 - \frac{1}{6}\dddot{v}(t)\left(\frac{\delta t}{2}\right)^3$$

We then subtract to get rid of the second derivative.

$$v\left(t+\frac{\delta t}{2}\right) - v\left(t-\frac{\delta t}{2}\right) = 2\dot{v}(t)\left(\frac{\delta t}{2}\right) + \frac{2}{6}\dddot{v}(t)\left(\frac{\delta t}{2}\right)^3 \tag{E.7}$$

We then do the Taylor series expansion for the *derivative*, $\dot{v}(t+\delta t/2)$ and $\dot{v}(t+\delta t/2)$, up to the $\dddot{v}(t)$ term:

$$\dot{v}\left(t+\frac{\delta t}{2}\right) = \dot{v}(t) + \ddot{v}(t)\left(\frac{\delta t}{2}\right) + \frac{1}{2}\dddot{v}(t)\left(\frac{\delta t}{2}\right)^2$$

$$\dot{v}\left(t-\frac{\delta t}{2}\right) = \dot{v}(t) - \ddot{v}(t)\left(\frac{\delta t}{2}\right) + \frac{1}{2}\dddot{v}(t)\left(\frac{\delta t}{2}\right)^2$$

Add to get rid of the second derivative:

$$\dot{v}\left(t+\frac{\delta t}{2}\right) + \dot{v}\left(t-\frac{\delta t}{2}\right) = 2\dot{v}(t) + \dddot{v}(t)\left(\frac{\delta t}{2}\right)^2$$

Multiply this by $(\delta t/6)$ and then subtract from equation (E.7):

$$v\left(t+\frac{\delta t}{2}\right) - v\left(t-\frac{\delta t}{2}\right) - \frac{\delta t}{6}\dot{v}\left(t+\frac{\delta t}{2}\right) - \frac{\delta t}{6}\dot{v}\left(t-\frac{\delta t}{2}\right) = 2\dot{v}(t)\left(\frac{\delta t}{2}\right) - \frac{2}{6}2\dot{v}(t)\delta t$$

The right side is just $2\dot{v}(t)\delta t/3$. Shift each t by δt and solve for $v(t+\delta t)$ to get:

$$\boldsymbol{v(t+\delta t) = v(t) + 1/6(\dot{v}(t+\delta t) + \dot{v}(t) + 4\dot{v}(t+\delta t/2))\delta t} \tag{E.8}$$

This is the **Simpson's rule method** of integration. It essentially fits the curve with a quadratic (rather than a linear line that the midpoint value method uses).

E.19: Should the value of the second derivative of v impact the accuracy of the Simpson's Rule method? Why or why not?

E.20: Should the value of the third derivative of v impact the accuracy of the Simpson's Rule method? Why or why not?

E.21: Add the Simpson's Rule method to the code you have for the rocket simulation and set the time step to 10^{-2} s. What value do you get? Do you find that the Simpson's Rule method (third-order approximation) is more accurate compared to the initial value (first-order), midrange, and midpoint (second-order) methods?

Problems

Problem E.1: If the acceleration is not constant, could the initial value method ever give a solution exactly equal to the analytical solution? If so, describe the situation. If not, why not?

Problem E.2: In our simulation, we assumed gravity and thrust were constant but the rocket mass decreased as the fuel was used up. Suppose we instead assume the rocket mass and thrust are constant but gravity decreases according to the universal law of gravitation: $F_g = Gm_1m_2/r^2$. Would the initial value method underestimate the velocity, overestimate the velocity, or neither? Provide your reasoning.

Problem E.3: Suppose we set the time step to be 10^{-16} s and then used the midpoint method to simulate the change in velocity that occurs with our rocket over a period equal to 10^{-13} s. Would this run into any issue with computer rounding? Why or why not?

Problem E.4: Suppose we wanted to numerically integrate a quadratic expression. Would any of our method(s) give us an answer exactly equal to the analytical value? If so, which ones and why? If not, why not?

Problem E.5: Which numerical method is being attempted in the following code, where `f` contains the function value and we are trying to find the integral of `f` from zero to 10? Are there any errors in it? If so, how would you fix it?

```
def f(x):
  return (x**3-1)*(x-1)**2+x**2

dx = 0.1
sum = 0
n = 100

for i in range(n):
  value = i*dx
  sum = sum + (f(value)+4*f*(value+0.5)+f*(value+1))*dx/6.
```

F. Differential Equations

- Physics context: Springs and water waves
- Programming skills: prompt, round, while loop, gvbars, .data, modulo
- Computational skills: Iterative methods
- Mathematics skills: Differential equations

F.1 Introduction to differential equations

F.1.1 Definition of a differential equation

Before I explain what a differential equation is, I want to discuss the physical situation we'll be examining in this part, namely the motion of a ball on a spring. The physics of this is rather straightforward.

An ideal spring exerts a force that is proportional to how far it is stretched or compressed. We can represent that force as $F = -kx$, where k is some constant value (called the **spring constant** or **spring stiffness**) and x is the spring's length relative to its natural length. The negative sign is there because the force is directed back toward the spring's "natural" length (where it exerts no force). We can express Newton's second law as $F = m\ddot{x}$, where I've indicated the acceleration as $\ddot{x}$ to simplify the notation (each ''dot" representing a derivative with respect to time). Replacing F in Newton's second law by $-kx$ gives:

$$-kx = m\ddot{x} \tag{F.1}$$

Equation (F.1) is a differential equation since it involves the variable x and the second derivative of x (the acceleration). A differential equation is an equation that involves both the variable and some derivative of it.

Our rocket problem of chapter E also results in a differential equation if we no longer assume a constant gravitational force and/or no drag because the gravitational force depends on the position of the rocket (how far it is

from the center of Earth) and the drag depends[i] upon the rocket's velocity. Including them therefore results in a differential equation, where the acceleration (second derivative of position with respect to time) depends upon the velocity (first derivative of position with respect to time) and the position.

Another common differential equation has to do with population growth. Suppose we have a group of ten animals and the group, as a whole, experiences two births and one death each year, for a growth rate of one per year.'That means $x = 10$ and $\dot{x} = 1/\text{yr}$.

Now let's suppose we had 100 animals. Each group of 10 experiences a growth of 1/yr so with 100 animals the growth would be 10/yr.

Notice how in both cases $x = (10yr)\dot{x}$. The population equation, $x = (10yr)\dot{x}$, is a differential equation.

F.1: According to the population equation, $x = (10yr)\dot{x}$, what is growth rate when the population is 1000?

F.1.2 Analytical solutions to differential equations

Obtaining an analytical solution to a differential equation can be a tricky process.

For our purposes, you don't need to do this for the ball and spring problem – I'll just provide you with the analytical solution. However, before I do so, let's look at some other differential equations and their analytical solutions so you can understand why the solution I provide makes sense.

Let's start with a general population equation, with a growth rate that is some fraction r of the population: $\dot{x} = rx$. The solution to this relationship is $x = Ae^{rt}$, where A is a constant and t is the time.

You can verify this is the solution by taking the derivative of x (with respect to t) and then plugging that back into the expression. In this case, the derivative of x equals Are^{rt}, which is equivalent to $r(Ae^{rt})$ or rx. Plug rx back into the expression for $\dot{x}$ (left side of expression) and you'll see that the two sides are equal.[ii]

---

[i] Drag also depends upon position since drag depends upon the air density and temperature both depend on altitude.

[ii] Note that $x = Ae^{rt}$ would also be the analytical solution to the equation $\ddot{x} = r^2x$.

As you can see, the solution to the population equation is an exponential function. It is for this reason that population growth tends to be exponential (until other limiting factors come into play).[iii]

F.2: Suppose we have a truck on a level road whose brakes have failed, such that there is no friction to slow it down. The only force acting to slow it down is air resistance, which we'll treat[iv] as proportional to the truck's speed ($F = -bv$). The equation of motion would therefore be $-bv = m\dot{v}$. Show that the analytical solution is as follows by plugging in the following expression into $-bv = m\dot{v}$ and showing that both sides are equal (note that I'm using "$\exp x$" to represent "e^x" so as to avoid over-crowding the exponent):

$$v = v_0 \exp\left(-\frac{b}{m}t\right)$$

F.1.3 Simple harmonic motion (SHM)

As mentioned earlier, we'll examine **oscillations**, as with a ball oscillating on a spring. A ball oscillating on a spring is simple enough that its motion can be solved analytically, which will allow us to check the accuracy of the numerical methods. In addition, if we can use the computer to model a ball oscillating on a spring, we can then examine more complicated situations like multiple balls all connected by springs (section F.2), which is much more complicated to solve analytically but rather straightforward to solve numerically.

As noted above, the equation of motion for a ball on a spring is provided by equation (F.1): $-kx = m\ddot{x}$. It turns out that the solution to this differential equation is as follows:[v]

$$x = A\cos\left(\sqrt{\frac{k}{m}}t + \phi_0\right) \tag{F.2}$$

[iii] I've been using x for the variable but we can use any letter for the variable, not just x. For example, it is traditional to use N (or sometimes P) to represent the population. For this reason, the population equation is typically written as $\dot{N} = rN$ with the solution being $N = Ae^{rt}$. Regardless of which letter we choose for the variable, you should be comfortable interpreting a differential equation and recognizing why the solution is what it is.

[iv] It is more likely proportional to the *square* of the speed.

[v] Notice that this is a sinusoidal function, consistent with a type of motion called **simple harmonic motion**.

To verify that this is, indeed, a solution, we need to take the second derivative of the cosine. Since the first derivative of the cosine is a negative sine, and the first derivative of the sine is a positive cosine, the *second* derivative of the cosine is a *negative* cosine. This is consistent with our spring equation of motion, which states that the second derivative $\ddot{x}$ is proportional to $-x$.

In our case, we'll keep one end of the spring fixed, as though it was attached to something like a ceiling, and attach a ball on the other end so that the ball hangs down due to gravity. That means there are two forces acting on the ball, not just one. In addition to the spring force (kx) there is also gravity (mg), so the equation of motion is as follows:

$$-ky - mg = m\ddot{y} \tag{F.3}$$

Notice that I am now using y instead of x as the variable for the position since the ball is oscillating vertically and it is the convention to use y to represent the vertical coordinate (with positive being upward). Also notice that the spring force is negative since the spring force will be upward when the spring itself is extended downward. The gravitational force is also negative since it is always directed downward.

This is a differential equation and can be solved analytically to get the following expression for y:

$$y = A\cos\left(\sqrt{\frac{k}{m}}t + \phi_0\right) - \frac{m}{k}g \tag{F.4}$$

This is the same as equation (F.2) except that I'm using y instead of x, and it has the added factor of $-mg/k$ because of the gravitational force. In our simulations, the ball starts at rest with the spring at its natural length, which means $\phi_0 = 0$ and so we have the following solution:

$$y = \frac{mg}{k}\cos\left(\sqrt{\frac{k}{m}}t\right) - \frac{m}{k}g \tag{F.5}$$

F.3: Verify that when the time is zero then the initial position y and the initial velocity $\dot{y}$ are both zero. You'll need to take the derivative of equation (F.5) with respect to time to get an expression for the velocity.

F.4: Verify that equation (F.5) is a solution to equation (F.3) in the same way you answered question F.2.

F.1.4 Using Python to plot the analytical solution

The Python program F:SHM uses the analytical result described earlier to calculate the position of an object undergoing simple harmonic motion so that it can plot the values during a period of 20 s. Note that it uses `hpos` to represent the vertical variable since `y` is reserved for VPython use (see section F.3) and using `y` for something else can potentially cause some confusion.

Note: This code uses the `prompt` function to allow the user to enter the time. The first parameter is the query the user sees and the second parameter is the "default" response. The code is set up so that if the user just hits "OK" (or enter) then the code interprets the answer as "20".[vi]

This code is not doing any numerical integration. It includes a `for` loop only for the purpose of plotting the value of `hpos` at every time during the 20 seconds. Note that the `for` loop goes to `nint+1` so that it can plot every point from `0` to `nint*dt`.

F.5: In the code, the `koverm` variable represents the value of k/m. Plug the value of k/m given by the code, along with a time of 20 s, into the equation for the position (equation F.5). What value do you get for the position of the object at 20 s? This is the analytical solution.

F.6: Run the program. What does the program predict is the position of the object 20 s after the start? This is also the analytical solution (since it uses the same equation you used in problem F.5). Given your result in problem F.5, does the value provided by the program make sense? Why or why not?

F.7: How does the oscillation change if `koverm` is larger? What is happening physically that would make the oscillation change in that way? In other words, what forces are responsible for the change and why?

F.1.5 Semi-implicit Euler or Euler-Cromer scheme

Now let's consider how we can get the computer to determine the position of a ball oscillating on a spring (suspended from the ceiling) without knowing

[vi]The code then uses the `float` function to convert the answer to a real number (see chapter D).

beforehand what the solution looks like. All we know is that the acceleration has the following form (rearranged from equation F.3):

$$\ddot{y} = -\left(\frac{k}{m}\right) y - g \tag{F.6}$$

This is easy enough to add to the code, as follows.

```
def accel(h):
  return 'koverm*h ' g
```

Since the object starts at rest and at position 0, we have:

```
h = 0.  # initial position
v = 0.  # initial velocity
```

Now we want to use the acceleration to predict the position. From chapter E, we know we can get the velocity by integrating the acceleration. Numerically, this means breaking up the time period into lots of little time steps and then adding up all the $a\ dt$'s, where a is the average acceleration during the time step. Likewise, we can get the position by adding up all the $v\ dt$'s, where v is the average velocity during the time step.

Programming these two steps in the same way we did in chapter E, it looks like the following:

```
for i in range(nint):
  v = v + accel(h)*dt
  h = h + v*dt
```

Add a new graph (you can copy the one already there that plots the analytic solution and just name it something different) and plot the values from the numerical simulation on the new graph. You'll need to insert a plot command inside the for loop.

Note: If your plot command inside the `for` loop is after you set the new `h`, you'll need to plot the first point (at time 0) before the `for` loop and the plot statement inside the loop needs to be at `t+dt`. If your plot command inside the `for` loop is before you set the new `h`, you'll need to plot the last point (at time `tf`) after for loop ends and the plot statement inside the loop needs to be at `t`.

F.8: Add the appropriate lines to the code to calculate the position numerically. Have the program plot the simulated position on a graph, and also print out the final position as determined by the integration. What is the final position?

F.1.6 Forward and Backward Euler schemes

You should find that the semi-implicit Euler method produces a simulation that is very similar to the analytical solution with almost identical graphs and similar but not the same ending positions (around −0.5 m).

Given that the velocity is determined using the acceleration at the beginning of each step, you might suspect (based on what we learned in chapter E) that this is causing an error, since the acceleration at the beginning of the step is not equal to the average acceleration during the step. The problem is that the acceleration depends upon the position, and so we don't know the acceleration at any other time except the beginning of the time step since that is all we know about the position.

However, we don't have the same issue with getting the position. After all, the first step gives us the velocity at the end of the time step. Consequently, it seems we can improve the second step by using the *average* of the two velocities – the one at the beginning of the time step and the one at the end of the time step.

This method is called the **Forward Euler scheme**. The code for this scheme looks like the following (remember to reset `v` and `h` to zero before the loop):

```
for i in range(nint):
  vnew = v + accel(h)*dt
  h = h + 0.5*(v+vnew)*dt
  v = vnew
```

Notice how the first line uses "`vnew =`" instead of "`v =`". This is because the next line needs to use both values of `v`, the value at the beginning of the time step and the value at the end.

F.9: Add the appropriate lines to the code to calculate the position via the Forward Euler scheme. Have the program create a new graph and also print out the final position as determined by the integration. What is the final position?

When you use the Forward Euler scheme to do the integration, you should find that the position becomes further and further off with time. Even though the position is calculated using a better estimate of the average velocity during the time step (average of velocity at beginning and velocity at end), the integration is off because the velocity is based off the acceleration at the

beginning of the time step rather than the average acceleration during the time step.

We encounter a similar issue if we decide to reverse the process and figure out the new position then determine the velocity, rather than the other way around. This method is called, not surprisingly, the **Backward Euler scheme**.

With the backward Euler scheme, we use the velocity at the beginning of the time step (rather than the midrange value) to determine the new position and then, since the new position is known, calculate the new velocity using the midrange of the beginning and end accelerations.

```
for i in range(nint):
  hnew = h + v*dt
  v = v + 0.5*(accel(h)+accel(hnew))*dt
  h = hnew
```

F.10: Add a new loop (and new plot) using the Backward Euler scheme (remember to reset `h` and `v` to the starting values before doing the loop). What value do you get? Is it similar to what you got with the Forward Euler scheme?

While the backward Euler scheme does a good job getting the velocity (because it uses the midrange of the initial and final accelerations during the time increment), the final position (and thus the final acceleration) is still off because it uses only the *initial* velocity to determine position. Consequently, the backward Euler scheme ends up being just as poor (or good) as the forward Euler scheme.

The reason the semi-implicit Euler method works better than either the forward or backward Euler methods is because the velocity and position predictions are likely to be off in *opposite* ways, leading to a more reliable prediction overall. In other words, the velocity prediction uses the acceleration at the *beginning* of the time step (rather than the average acceleration) and the position prediction uses the velocity at the *end* of the time step (rather than the average velocity). Neither is correct separately but together the errors counteract and produce a good prediction of the position.

F.11: Suppose you compared the velocity values produced by the semi-implicit Euler scheme with the analytical values. Do you think there would be as good a match as there is with the position values? Why or why not?

F.1.7 Runge-Kutta or Heun scheme

Even though the semi-implicit Euler scheme seems pretty good, it is probably not as popular as the Runge-Kutta (or **Heun**) scheme.

In the **Runge-Kutta scheme**, we start with the forward Euler scheme (meaning we use the acceleration valid at the beginning of the time increment to figure out the change in velocity and then use the midrange of the old and new velocities to get the new position):

```
for i in range(nint):
    vnew = v + accel(h)*dt
    hnew = h + 0.5*(v+vnew)*dt
```

However, unlike the forward Euler scheme, we add one more step before moving on. In this *extra* step we use the midrange of the old and new accelerations (since we now know both the old and new positions) to get a new, updated prediction of the velocity at the end of the time increment:

```
    v = v + 0.5*(accel(h)+accel(hnew))*dt
    h = hnew
```

In this way, the velocity value is improved for use during the next time step.

F.12: Add a new loop (and new plot) using the Runge-Kutta scheme (remember to reset `h` and `v` to the starting values before doing the loop). What value do you get?

F.13: How does the Runge-Kutta scheme compare to what you obtained with the other schemes so far?

F.1.8 Leapfrog scheme

There is one final scheme we will examine. This scheme is called the **leapfrog scheme**, for reasons that will become obvious shortly.

In the leapfrog scheme, we first integrate to find the velocity *halfway* through the first time step (for a total time step equal to $\delta t/2$):

```
    v = v + accel(h)*dt/2
```

Next, we use this "middle" value to predict the position at the *end* of the time step (for a total time step equal to δt):

```
h = h + v*dt
```

At this point, we have the velocity *halfway* through the first time step, and we have the position at the *end* of the first time step. We can then use the acceleration at the *end* of the first time step to "leapfrog" over the end of the time step and predict the velocity halfway through the *next* time step (for a total time step equal to δt):

```
v = v + accel(h)*dt
```

In this way, we continue to predict the velocity and the position at *different* times, with the velocity being predicted at "half-way" points, which is why this is called the "leapfrog" method. In fact, the last two lines above can be put inside a loop as follows:

```
for i in range(nint):
    h = h + v*dt
    v = v + accel(h)*dt
```

At each time step, we are doing two calculations, one for position and one for velocity, but their values are valid at *different* times. The new position is valid at the end of the time step while the new velocity is valid at the middle of the next time step.

F.14: Add a new loop (and new plot) using the leapfrog method (remember to reset `h` and `v` to the starting values before doing the loop). What value do you get? Also provide a link to your code containing all five methods (forward Euler, backward Euler, Semi-implicit Euler, Runge-Kutta, and leapfrog). Make sure your code is in a "Public" folder so that it is accessible by others.

In this section, we explored a situation (a single spring) for which an analytical solution was readily available, as that allowed us to see how accurate the various numerical methods were. Having multiple objects, all connected by springs,[vii] is much more difficult to solve analytically, whereas the numerical approach for a single object can easily be extended to multiple objects. This is shown in section F.3.

[vii] To see what I mean, check out the PhET simulation wave-on-a-string. Playing with the simulation will give you an idea of where we are going, although our approach will be different – whereas PhET uses an analytical solution (only possible because they constrain the balls to only move up and down), we will use a numerical technique (thus allowing the balls to move in all three dimensions).

F.2 Partial differential equations

F.2.1 The water wave model

In section F.1, we used computational techniques to simulate the time evolution of a scenario governed by a differential equation. In this section, we examine doing the same thing for a scenario governed by a *partial* differential equation. A simple illustration of this is provided in the water-wave program.

Run the program. The bar graph represents a water surface. You should notice that the water surface starts off with a "pile" of water near the left edge that evolves into two separate wave pulses that move off to the left and right, consistent with the way we expect water to behave.

F.15: When you run the code, what happens when the wave pulses encounter the left and right sides of the domain? Do they bounce off (reversing directions) or do they continue to cycle around, with the pulse leaving the right edge simultaneously entering from the left edge?

F.16: You should notice that the wave pulses do not continue to travel across the water surface forever. Instead, the simulation "blows up". How many times does the wave pulse run the length of the domain before the program blows up?

F.2.2 The physics

As mentioned in Question F.16, the simulation "blows up", we'll examine a couple of ways we can use computational techniques to improve the simulation. Before getting to that, however, it is important that you understand the code. While it isn't necessary for you to fully understand the physics behind the equations used in the code, a little bit of background is helpful for understanding why the integration scheme involves a partial differential equation and how we will "fix" the code. The complete derivation is provided in section F.4.

Just as we determine the position of an object by integrating its velocity, the height (vertical position) of the water surface is determined by integrating the vertical velocity of the surface (upward velocity means the increasing

height). The integration involves a loop with the following statement (written in mathematical symbols here rather than Python variables):

$$h(x, t + \delta t) = h(x, t) + \frac{\partial h(x, t)}{\partial t}\delta t \tag{F.7}$$

In this expression $h(x, t)$ represents the height of the water at each point. The "(x, t)" group means that the h value depends on the horizontal position x and time t. As in chapter E, the new value of h acts as the "initial state" for the next iteration and the process is repeated, over and over again.

Important note about the limitations of this simulation: Real water moves both vertically and horizontally but our water surface can only move vertically (which is why we can use a bar graph[viii] to represent it). Since our water surface only has a single height at each point, the program can't be used to simulate wave breaking (as when the waves "tip over" when approaching a beach). Despite this limitation, it represents basic characteristics of water waves and provides a context for investigating how we deal with partial differential equations.[ix]

With the spring (section F.1), we needed to integrate twice, integrating the acceleration to get the velocity and then integrating the velocity to get the position. With the water wave, we also need to integrate twice, not because we have a second derivative (acceleration) but rather because we have a partial differential equation.

It turns out that the vertical velocity of the water surface ($\partial h/\partial t$) depends on four things: (1) the height of the water h, (2) how that height varies across the surface $\partial h/\partial x$, (3) how fast the water is moving horizontally u, and (4) how that speed varies across the surface $\partial u/\partial x$. Some notes:

- I'm indicating the height of the water surface as h, which is equivalent to $h(x, t)$, as it depends on both the horizontal position x and the time t.
- Similarly, the horizontal velocity of the water, u, is equivalent to $u(x, t)$, and u is the *rightward* motion of the water, so it is positive when the water moves rightward.

[viii] In other words, the bars themselves are not moving leftward or rightward. The bars just represent the height of the water due to the water moving leftward or rightward.

[ix] Simulations here, here and here allow breaking by modeling the water as particles that are free to move in any direction.

The actual relationship between the vertical velocity $\partial h/\partial t$ (used in equation F.7) and the four things listed above is as follows (see derivation for why):

$$\frac{\partial h}{\partial t} = -h\frac{\partial u}{\partial x} - \frac{u\partial h}{\partial x} \tag{F.8}$$

This is called a **partial differential equation** because it includes partial derivatives. A **partial derivative** is one that is taken respect to one variable while the other variable is held constant. For example, consider water flowing down a river. If we measure the rate at which the temperature of the water is changing at a particular location, then we are measuring $\partial T/\partial t$, where T is the temperature, t is the time, and the position x remains constant. Conversely, if we take a snapshot of the temperature at a particular time, we can examine the gradient of the temperature along the river, which would be $\partial T/\partial x$.

Expression (F.8) should make sense. Let's look at each term on the right side:

- The first term on the right side is looking at whether the water is moving into the location or out of it. Since u is the horizontal motion of the water, then $\partial u/\partial x$ is positive when more water is leaving than entering. Thus, a positive value corresponds to a lowering of the water height and that is why there is a negative sign for this term.
- The second term on the right side is looking at whether water from a higher location is being pushed into the location. A positive $\partial h/\partial x$ means that the water to the right is higher and water to the left is lower. A positive u means water is moving to the right, bringing in the lower height water. This corresponds to a lowering of the water height and that is why there is a negative sign for this term as well.

Not only does h change over time but u does as well, which means there is a similar expression governing the change in u:

$$u(x, t + \delta t) = u(x, t) + \frac{\partial u(x, t)}{\partial t}\delta t \tag{F.9}$$

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - g\frac{\partial h}{\partial x} \tag{F.10}$$

Equations (F.9) and (F.10) look like equations (F.7) and (F.8), but they are not the same. Equation (F.9) is used to integrate $\partial u/\partial t$ to get u, whereas

equation (F.7) is used to integrate $\partial h/\partial t$ to get h. And, while both equations (F.8) and (F.10) use $\partial h/\partial x$ and $\partial u/\partial x$, they use them in different ways. Let's look at each term in equation (F.10):

- The first term on the right side of equation (F.10) is like the second term on the right side of equation (F.8) in that it is looking at whether water from a faster location is being pushed into the location. A positive $\partial u/\partial x$ means that the water to the right is faster (rightward) and water to the left is slower (or faster leftward). A positive u means water is moving to the right, bringing in the slower water. This corresponds to a slowing of the water and that is why there is a negative sign for this term.
- The second term on the right side of equation (F.10) is looking at whether the gravity is accelerating the water one way or the other. A positive slope to the water height will mean higher pressure to the right, pushing the water leftward. Since u is the rightward motion of the water, then a positive $\partial h/\partial x$ corresponds to a deceleration of u and that is why there is a negative sign for this term.

F.17: Find the lines within the code that represent equations (F.8) and (F.10). What names are being used for u, $\partial h/\partial x$ and $\partial u/\partial x$ variables in the Python code?

F.2.3 The code

As you read through the code, you'll notice that I mostly use techniques introduced in previous parts. For example, I use the `.append` method to add values to an array.

There are a couple of new things, though. Starting from the top, the first thing that is new is the way the program does the plot. We have done curves before but this is the first time we have done a bar graph. The initial display is similar but this uses `gvbars` instead of `gcurve`. The program calls the plot `waveplot` but that name is arbitrary.

Rather than plot each point using `.plot`, as in `waveplot.plot(x,y)`, the program uses `.data`, as in `waveplot.data=(xypairs)`, where `xypairs` is an array that contains two values, the x and y coordinates, for each point to be graphed. In this program, the array is called `hplot`, which is initialized on line 26, with initial values assigned on line 54 (two values to each element

in the array), and the values first plotted on line 56. Within the integration loop, the y-component of each element in the array is updated on line 83 and the graph is replotted each time through the loop on line 85.

The program uses `xmid` to indicate where the initial pile of water is along the x axis. Notice how the program figures out how it determines the value of `xmid`.

F.18: How would you change the program so that the initial pile of water is closer to the center rather than the edge? Explain why that change does what you want.

The bottom of the code is similar to what you've seen before, with the height and horizontal speed at each time determined by integrating the time derivatives of each, as in equations (F.7) and (F.9):

```
for i in range(N):
  h[i] = h[i] + dhdt[i] * dt
  u[i] = u[i] + dudt[i] * dt
```

Just before this, the program calculates the time derivatives, using equations (F.8) and (F.10):

```
for i in range(N):
  dhdt[i] = -h[i] * dudx[i] - u[i] * dhdx[i]+ k * d2hdx2[i]
  dudt[i] = -u[i] * dudx[i] - g * dhdx[i]
```

Before this, in the first for loop, the program determines the derivatives $\partial h/\partial x$ and $\partial u/\partial x$ (coded as `dudx` and `dhdx`, respectively) using the symmetric difference (second order) approach from chapter A:

```
for i in range(N-1):
  dhdx[i] = ( h[i+1] - h[i-1] ) / (2 * Deltax)
  dudx[i] = ( u[i+1] - u[i-1] ) / (2 * Deltax)
```

Note that, unlike the other loops, this `for` loop uses `range(N-1)` not `range(N)`. That means the loop only covers `N-1` values, not all `N` values, and stops before the last point (the one on the far right). Because the loop doesn't assign the last point, there are two extra lines after the loop that assign the values at that last point:

```
dhdx[N-1] = ( h[0] - h[N-2] ) / (2 * Deltax)
dudx[N-1] = ( u[0] - u[N-2] ) / (2 * Deltax)
```

To understand why the code does this, one needs to first recognize how Python refers to each part of an array. As mentioned above, the first element

in an array of size `N` is at index `0`, the next is at index `1`, and so on up to index `N-1`. For example, for array `dhdx` of size `N`, the elements are `dhdx[0]`, `dhdx[1]`, `dhdx[2]` and so on up to `dhdx[N-2]` and `dhdx[N-1]`. Notice how the last element is at `N-1`, not `N`, since the array starts with element `0` and we need to have `N` elements overall. Notice also how there is no `dhdx[-1]` element and no `dhdx[N]` element. Those elements simply don't exist.

F.19: If there are `N` elements in the array, why is it said that `dhdx[N]` doesn't exist?

We can now explain why the for loop only covers `N-1` values instead of `N` values. To calculate the derivative at location `i`, the program uses the value at the location to the left (`i-1`) and the value at the location to the right (`i+1`). For the last location on the right, the index `i` is equal to `N-1`, which means it would be trying to use the values at indices `N-2` and `N`. There is no value at index `N` so any attempt to access that would cause an error. This is why separate lines are needed to calculate the derivative there. For cyclical boundaries (as you identified earlier in this section), we can use the extra lines to specify that we want to use the value at index `0` instead of the value at index `N`.[x]

F.20: Why are cyclic boundaries consistent with using the value at index `0` for the derivative at `N-1` instead of the value at index `N`?

Hopefully at this point you are now wondering about the *left* end. In particular, if the *right* end (at index `N-1`) needs to be separated from the loop (so that it doesn't try to access index `N`, why doesn't the *left* end (at index `0`) likewise need to be separated from the loop (so that it doesn't try to access index `-1`)? The answer is that Python "knows" that there can't be negative indices so it automatically assumes we mean to subtract from the "other" end of the array.

F.21: Via the `range(N-1)` specification, the code uses a loop for indices from `0` to `N-2` to calculate the derivatives. Suppose we replace the loop with the following (keeping the two lines following the loop). Would that also work? Why or why not?

[x] Instead of using cyclical boundaries, we could instead have the code use a "forward 2-pt" scheme for the first point and a "backward 2-pt" scheme for the last point.

```
dhdx[0] = ( h[1] - h[N-1] ) / (2 * Deltax)
dudx[0] = ( u[1] - u[N-1] ) / (2 * Deltax)
for i in range(1,N-1):
  dhdx[i] = ( h[i+1] - h[i-1] ) / (2 * Deltax)
  dudx[i] = ( u[i+1] - u[i-1] ) / (2 * Deltax)
```

F.2.4 Simplifying the code

Rather than separate out the calculation at index `N`, we can instead use the **modulo operator**. The modulo operator determines the **remainder**. For example, if you divide 14 by 4, the remainder is 2, since 4 goes evenly into 12 with 2 "remaining" to get to 14. In Python, this is written as `14%4`, which is equivalent to 2.

For our code, we could use `(i+1)%N`, instead of `(i+1)` in the loop. When `i` is less than `N-1`, this would be equivalent to `i+1`. When `i` is equal to `N-1`, this would be equal to the remainder of `N` divided by `N`, which is equivalent to `0`.

Make a copy of the code for you to edit, and then replace the following five lines. . .

```
for i in range(N-1):
  dhdx[i] = ( h[i+1] - h[i-1] ) / (2 * Deltax)
  dudx[i] = ( u[i+1] - u[i-1] ) / (2 * Deltax)
dhdx[N-1] = ( h[0] - h[N-2] ) / (2 * Deltax)
dudx[N-1] = ( u[0] - u[N-2] ) / (2 * Deltax)
```

. . . with these three:

```
for i in range(N):
  dhdx[i] = ( h[(i+1)%N] - h[i-1] ) / (2 * Deltax)
  dudx[i] = ( u[(i+1)%N] - u[i-1] ) / (2 * Deltax)
```

This change shouldn't impact the simulation, but it makes it easier to code, which will make it easier to implement the fixes that will be discussed later.

F.22: The modification had you change the loop to `range(N)` and replace `i+1` with `(i+1)%N`. Could you also replace `(i-1)` with `(i-1)%N`? Why or why not?

F.2.5 Improving the first derivative

To address the problem of the code blowing up, we'll apply some of the techniques we've learned about in the course.

The first thing we'll try is to improve the calculation of the first derivative. As we know from chapter A, the second-order (three-point) method is just one way of calculating the derivative. A more accurate method is the fourth-order (five-point) method:

$$f'(x) = \frac{f(x-2\delta x) - 8f(x-\delta x) + 8f(x+\delta x) - f(x+2\delta x)}{12\delta x} \quad \text{[fourth order]}$$

Edit the water wave program to implement the fourth-order scheme instead of the second-order scheme.

F.23: With the fourth-order approximation to the first derivative, how many times does the wave pulse run the length of the domain before the program blows up?

F.2.6 Modifying the integration scheme

While the fourth-order approximation is more accurate than the second-order approximation, making that replacement doesn't improve the wave simulation. That suggests the problem isn't with the accuracy of the derivative.

We know from section F.1 that there are a variety of integration schemes. As we know, the ideal scheme uses the average time derivative during each time step. In this case, the code is using the beginning time derivative during each time step, as the code first calculates all the space derivatives (valid at the beginning of the time step) and the time derivatives (valid at the beginning of the time step) before doing the integration.

Note: The spatial derivative, $\partial/\partial x$, is called the **gradient**. To simplify the language, I'll refer to the spatial derivative as the gradient.

We can improve the scheme by using the semi-implicit approach. With the semi-implicit approach, we integrate the heights h using the values at the beginning of the time step, as before, and then, after updating the heights and recalculating the height gradients $\partial h/\partial x$, we integrate the velocity u using the updated heights and height gradients.

Note: The time derivative of u depends on the value of u, and the code only knows those values at the beginning of the time step, so the time derivative isn't using *everything* valid at the end of the time step.

The semi-implicit scheme can be written with the second two loops reorganized, one for h and one for u, as follows.

```
for i in range(N):
  dhdx[i] = ( h[(i+1)%N] - h[i-1] ) / (2 * Deltax)
  dudx[i] = ( u[(i+1)%N] - u[i-1] ) / (2 * Deltax)
for i in range(N):
  dhdt[i] = -h[i] * dudx[i] - u[i] * dhdx[i]
  h[i] = h[i] + dhdt[i] * dt
  hplot[i][1] = h[i]
for i in range(N):
  dhdx[i] = ( h[(i+1)%N] - h[i-1] ) / (2 * Deltax)
  dudt[i] = -u[i] * dudx[i] - g * dhdx[i]
  u[i] = u[i] + dudt[i] * dt
```

F.24: Why is the `dhdx[i]` calculation in two loops rather than just one or the other?

F.25: Implement the semi-implicit scheme. This should improve the simulation. Does it improve it significantly, or just a little bit?

F.2.7 Modifying the integration scheme, part 2

Given that the semi-implicit scheme works better, one might wonder if it would be sufficient to combine all three loops into a single loop, as follows.

```
for i in range(N):
  dhdx[i] = ( h[(i+1)%N] - h[i-1] ) / (2 * Deltax)
  dudx[i] = ( u[(i+1)%N] - u[i-1] ) / (2 * Deltax)
  dhdt[i] = -h[i] * dudx[i] - u[i] * dhdx[i]
  h[i] = h[i] + dhdt[i] * dt
  hplot[i][1] = h[i]
  dudt[i] = -u[i] * dudx[i] - g * dhdx[i]
  u[i] = u[i] + dudt[i] * dt
```

Doing this will result in a simulation that is similar to that for the semi-implicit scheme but it results in an asymmetry, as the code is calculating

gradients using updated values on the left end but not on the right end. If you implement this in the code, you should see a slight asymmetry in the water waves. For that reason, the two-loop method is better.

F.2.8 Viscosity

To make the simulation more realistic, we can add **viscosity**. When different parts of a fluid move at different speeds, there is a sort of friction that acts to slows down the faster portion and speeds up the slower portion. Due to viscosity, a fluid exerts a force on itself that is proportional to the shear (the gradient in velocity). The greater the shear, the greater the force back toward uniformity.

To understand how we'll add viscosity to the model, recall that for our one-dimensional water wave, the equation for the time derivative of h (height of water surface) is as follows:

$$\frac{\partial h}{\partial t} = -h\frac{\partial u}{\partial x} - u\frac{\partial h}{\partial x}$$

We want to add a **third** term that acts like viscosity to force the fluid toward uniformity. It turns out that a second derivative will do the trick (indicated in blue):

$$\frac{\partial h}{\partial t} = -h\frac{\partial u}{\partial x} - u\frac{\partial h}{\partial x} + k\frac{\partial^2 h}{\partial x^2} \tag{F.11}$$

Consider, for example, a location where the water surface is much higher or lower than either of the two adjacent points. The slope at that point can be small but the SECOND derivative is large. When we add a term that is proportional to the second derivative, it will be negative for high peaks, acting to bring the peaks down, and positive for low troughs, acting to bring those troughs up.

From chapter A (section A.5), we know that the second derivative can be determined as follows (first is the third-order or 3-point scheme and the second is the fifth-order or 5-point scheme):

$$f''(x) = \frac{f(x-\delta x) - 2f(x) + f(x+\delta x)}{(\delta x)^2}$$

$$f''(x) = \frac{-f(x-2\delta x) + 16f(x-\delta x) - 30f(x) + 16f(x+\delta x) - f(x+2\delta x)}{12(\delta x)^2}$$

Add viscosity to your code (which approximation you use for the second derivative is up to you). Remember to initialize your new second derivative variable, like `d2hdx2`, along with a viscosity coefficient (indicated as k in equation F.11; a value of 0.005 m^2/s seems to work well for the simulation[xi]). Calculate[xii] the second derivative within the loop, multiply it by the coefficient and then add the product to the expression for `dhdt`. Once you get it working, try out a couple of coefficient values and see which one you think works best. Larger coefficients correspond to thicker (and more viscous) fluids.[xiii]

F.26: Does the addition of viscosity improve the simulation? If so, how?

F.3 Using VPython with Multiple Springs

F.3.1 Introduction to VPython

In section F.1, we explored a situation (a single spring) for which an analytical solution was readily available. This allowed us to see how accurate the various numerical methods were.

However, what if we have multiple objects, all connected by springs?[xiv] That is much more difficult to solve analytically, whereas the numerical approach for a single object can easily be extended to multiple objects.

To show the evolution, the code will utilize **VPython**, which is just like Python but with extra commands designed to create visual objects, like spheres, blocks, arrows and so forth. The program F:SHM-V provides the

[xi]The kinematic viscosity of water is on the order of 10^{-6} m^2/s so using 0.005 m^2/s is more like tar. I'm not sure why that is so far off.

[xii]To square something in Python, you can use `Deltax**2` or `Deltax*Deltax`. In Python the carat (ˆ) means something else (comparing the two numbers bit by bit).

[xiii]If we add viscosity to the code that blows up, it no longer blows up. However, TOO MUCH viscosity makes it blow up even quicker, possibly because the viscosity pushes the water so much the opposite way that it creates individual extreme peaks.

[xiv]To see what I mean, check out the PhET simulation wave-on-a-string). Playing with the simulation will give you an idea of where we are going, although our approach will be different – whereas PhET uses an analytical solution (only possible because they constrain the balls to only move up and down), we will use a numerical technique (thus allowing the balls to move in all three dimensions).

VPython version of what was provided in section F.1 (except that it integrates forward in time indefinitely rather than stopping after a particular amount of time).

When you run the code, you should see two ball and spring combinations. The position of the ball on the left is determined analytically while the position of the ball on the right is determined numerically using the leapfrog scheme. You should find that the numerical solution matches the analytical solution.

Note: If you were instead running VPython from your computer then you'd need to add a line that tells the code to import the visual library (this is done via a `from visual import *` line). We do not need to include the import line since we are using glowscript.

There are several differences between this code and the code we had before.

- Since we are allowing the spring to move in all three dimensions (potentially), `g` is set as a three-dimensional vector with components (g_x, g_y, g_z). This is done by using the `vector` function. In particular, we want `g` to be `vector(0,-9.8,0)` so that `g` corresponds to a downward acceleration of 9.8 m/s2. This means that the analytical function `hpos` uses `mag(g)` because `g` is a vector in this program, not a scalar, and the `accel` function uses `+g` not `-g`.
- When using visual objects, each object needs to be set up. In this case, we have two ball-and-spring sets: one corresponding to the analytical solution and one corresponding to the numerical solution.
- Each object is given a name. In this case, I'm using `ball` and `spring` for the ball and spring that is being numerically simulated, and I'm using `ballref` and `springref` for the ball and spring that is representing the analytical solution. Characteristics of each object, like its position and velocity, are represented by adding a "suffix" to the name, like `ball.position` for the position of the ball.
  Note: The code initializes the radius and color of the ball within the `sphere` call, whereas it specifies the position and velocity of the ball in separate lines. We could have, instead, specified the ball's initial position and velocity within the `sphere` call. And, conversely, we could have specified something like the ball's radius in a separate line, as opposed to within the `sphere` call.
  It is easy to see that the ball's velocity is set to zero (starts at rest) but

it seems strange to set the ball's position as `springlen+rball`. This is because the spring is hanging from a point coincident with the origin and, since the spring length is equal to `springlen`, that means the ball is initially a distance `springlen` below the origin. In addition, since it isn't the center of the ball that is connected to the spring but the edge of the ball, we add `rball` (the radius of the ball).

- VPython will automatically station the camera (the user's position) facing the origin. However, that view may not be the best for a given simulation. In this case, we don't want the camera to face the origin, since that is only one end of the spring, and not the end where the ball is placed. Instead, we want it to face a location that is *below* the initial location of the ball since the ball is going to fall as soon as we release it (due to gravity). How far below will depend on how stiff your spring is. I've found that when `koverm` is 5, the following line sets the camera at a good location. If you use a smaller `koverm` then your spring is going to stretch farther and you'll need to lower your camera location.
  ```
  scene.center=vector(0,-springlen-1,1)
  ```
  By default, the camera will automatically move away from the objects when it determines it is necessary to do so (to keep all the objects in the frame of view). We'd rather it not do that, which is why I added the following line.
  ```
  scene.autoscale=False
  ```
  These two lines do not impact the simulation. They only impact what we see, so you can certainly change these if you want.
- This code uses the `scene.pause()` command to pause the simulation, allowing the user to start it whenever it is convenient to do so.
- This code uses a `while` statement with condition `True`. This allows the simulation to run until the user decides to close the window (or change the window), rather than stopping at a predetermined time, like 20 seconds.
- The code includes a `rate` command. This statement tells VPython to wait `dt` seconds before continuing. VPython requires a rate line in any loop that involves animation because VPython needs time to revise the picture. Without this, VPython could be going so fast through the loop that the graphics can't keep up (the rate line does nothing if the calculation takes longer than is needed to update the picture). By using `1/dt` instead of a specific number like `100` (where the program would wait $1/100^{\text{th}}$ of a second) the program should represent the motion of

the ball in a realistic time frame (assuming the computer itself can keep up).

- The integration part is identical to what we had in section F.1. The only tricky part is specifying the position of the ball and calculating the acceleration based upon that position. Since it is in three dimensions, some additional bit of coding is needed. For example, to figure out the position to use for the `accel` function, I subtract out `norm(spring.axis)*springlen` from the spring's length. The `norm` function normalizes a vector, which means that `norm(spring.axis)` is just a unit vector with direction equal to the spring's orientation. This gives the amount and direction of spring is stretched (or compressed) beyond its equilibrium position. And, whenever the ball's position is changed, we must also change the spring, so that it stretches to wherever the ball is.

F.3.2 Allowing for a ball off-axis

To illustrate the power of the numerical approach, consider that we can easily modify the program start with the ball off-axis. This can be done by simply having a non-zero value for the x component of the ball's initial position:

```
ball.pos=vector(1,-springlen-rball,0)
```

Make a copy of the program for yourself, make the edit described above and run the code. You should find that the ball swings back and forth as well as up and down.

You should also notice that the simulated ball no longer matches the analytic solution. This is because the analytic solution was for a very specific case, with the ball not off-axis.

F.3.3 Multiple objects

The numerical process is the same regardless of how many balls we have. However, the code is more complicated for two main reasons: (1) we need to keep track of multiple balls, not just one, and (2) each ball has an additional force acting on it due to each ball having two springs attached to it, not just one.

The F:SHM-N link is to a VPython program that handles multiple balls connected by springs.

When you run the code, you should see a series of five balls connected by springs, initially oriented at a slight angle, that oscillate in a somewhat chaotic manner.

These changes from the previous program are listed below in the same order that they appear in the program so you can follow along.

- We don't have an analytical solution to this situation, so there is no `hpos` function or comparison set of five balls.
- The acceleration function has changed for every ball except the bottom one, in that there is now an extra force on each of the other balls – that due to the spring attached to the ball below it.
- There is an additional line (23) specifying the number of balls as `N`. In the code I set `N` to be 5 but you can set it to whatever you'd like. Just keep in mind that the more balls the harder the computer needs to work, since it needs to do the numerical integration for each ball.
- The ball radius (`rball`) now depends on the number of balls (line 28). This is because the more balls that are present the farther the camera needs to be (see line 49) to get all the balls in the frame, which means the balls need to be larger so that they can seen.
- The balls and springs are now arrays, so they need to be initialized as arrays (lines 32-33; see chapter B for information about arrays):

```
ball = []
spring = []
```

- There are two loops (lines 34-38 and lines 42-45) that set up the characteristics of each ball and spring. The way the code does this is "append" each object to the array. There are two ways to do this and, just to illustrate the two ways, the program uses one way for the ball and the other way for the spring. You can, of course, use either method for each object.[xv]

```
ball.append(sphere(radius=rball, color=vector(0,1,0), \
  velocity=vector(0,0,0), \
  pos=vector(i,-(i+1)*springlen-(2*i+1)*rball,0)))
spring = spring + helix(coils=10, radius = N*0.2, \
  color=vector(1,0.5,0), thickness = N*0.04)
```

[xv] Note that specifications like the position and velocity are included in the sphere statement rather than separately (as in the previous program).

Both methods create a series of elements in an array, and you can refer to a particular ball or spring by using square brackets (for example, `ball[0]` and `ball[1]` would refer to the first and second elements of the `ball` array, respectively).

- When setting the spring positions and orientations, the first spring is done separately (before the loop) because it is only connected to one ball (the others have a ball at each end).
- Since there are multiple balls, there is a loop that integrates the velocity to get the position (lines 71-72). There is also a loop to update the spring axis and orientation (lines 76-81), with the first spring done separately (before the loop) because it is connected to just one ball rather than two. Similarly, there is a loop to integrate the acceleration to get the velocity (lines 85-88), with the last ball done separately (after the loop; lines 92-93) because it is connected to just one spring rather than two.

Note: Our code doesn't prevent the balls from "passing through" each other. If we really wanted them to "bounce" off each other, we'd have to add that to our code.

F.4 Derivation of Water Wave Equations

F.4.1 Basic Assumptions

The real physics of numerical modeling comes in deriving the equations that govern the time evolution of our system. For our one-dimensional water wave, those provide the time derivatives of h (height of water) and u (horizontal motion of the water):

$$\frac{\partial h}{\partial t} = -h\frac{\partial u}{\partial x} - u\frac{\partial h}{\partial x} \tag{F.8}$$

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - g\frac{\partial h}{\partial x} \tag{F.10}$$

To start with, we'll make the following two assumptions.

1. First, we'll assume that the water is **incompressible** (i.e., the density at each point doesn't change):

$$\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z} = 0 \tag{F.12}$$

The horizontal and vertical speeds are u and w respectively. Our horizontal and vertical coordinates are x and z respectively. So, x is the distance from, say, the left-most grid point in our domain and z is the height above the lowest grid point in our domain.[xvi]

2. Second, we'll assume that the water is in **hydrostatic balance**. In other words, the pressure gradient (pushing the water up) is equal to the gravitational force (per volume) pulling the water down:

$$P = P_{\text{top}} + \rho g(h - z) \tag{F.13}$$

where P is the pressure at a particular level and P_{top} is the pressure at the top of the water (equal to atmospheric pressure). The other variables are ρ, which is the average density from the top of the water to the particular level, and g (which we can assume is relatively constant at 9.8 N/kg).

F.4.2 Time Derivative of h ($\partial h/\partial t$)

Notice that I am treating the water as being two dimensional, not one-dimensional. The variables of motion (u and w) can vary with both horizontal position (x) and height or depth (z).

To make it one-dimensional, we'll eventually get rid of the vertical motion variable w by relating the vertical motion with the vertical motion of the water surface h (which is the height of the water surface above $z = 0$, wherever we happen to set $z = 0$). However, first we'll get expressions for the time-derivatives of h, u and w.

The first expression we'll derive is the one for the time-derivative of h, $\partial h/\partial t$, which is the vertical velocity of the water surface. Via calculus, the partial derivative can be related to the total derivative as follows:footnoteTo be complete, $-w\partial h/\partial z$ should be included but since h does not depend on z, that term is necessarily zero.

$$\frac{\partial h}{\partial t} = \frac{dh}{dt} - u\frac{\partial h}{\partial x}$$

[xvi]We could, instead, make x the distance from the middle of our domain and z the height above the mean ocean surface. The choice of zero is arbitrary.

The difference between $\partial h/\partial t$ and dh/dt is that the former represents the vertical motion of the water at a particular location (constant x and z) whereas the latter represents the vertical motion of the water as we follow the water motion (which may be moving horizontally as well as vertically). It is the latter (dh/dt) that is equivalent to w_{top}, the vertical motion at the top. Consequently, we have

$$\frac{\partial h}{\partial t} = w_{\text{top}} - u\frac{\partial h}{\partial x} \tag{F.14}$$

F.4.3 Time Derivatives of u ($\partial u/\partial t$) and w ($\partial w/\partial t$)

As we did with the height h, the total time derivatives of u and w can be related to the partial time derivatives as follows:

$$\frac{\partial u}{\partial t} = \frac{du}{dt} - u\frac{\partial u}{\partial x} - w\frac{\partial u}{\partial z}$$
$$\frac{\partial w}{\partial t} = \frac{dw}{dt} - u\frac{\partial w}{\partial x} - w\frac{\partial w}{\partial z}$$

As with the expression for $\partial h/\partial t$, we need to replace the total derivatives with something we can calculate in the model. To do this, we use Newton's second law.

Newton's second law equates the net force per mass acting on an object with the acceleration of the object. For us, the water at each point can be accelerating in two directions, x and z. The acceleration in the x direction is du/dt and the acceleration in the vertical direction is dw/dt.

For the water at a particular point, the force on it is due to pressure differences in the water. If the pressure on one side is greater than the pressure on the other side, the water at that location will accelerate toward the lower pressure side.

This force per mass is related to the pressure gradient. The horizontal pressure gradient force is equal to $-(1/\rho)(\partial P/\partial x)$ and the vertical pressure gradient force is equal to $-(1/\rho)(\partial P/\partial z)$.

There is also the force per mass due to gravity (equal to g). As discussed above, we'll assume that exactly balances out the vertical pressure gradient force (hydrostatic balance). Consequently, the net vertical force will be zero.

When we apply Newton's second law to the water, we can replace du/dt and dw/dt by $-(1/\rho)(\partial P/\partial x)$ and zero, respectively, to get

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - w\frac{\partial u}{\partial z} - \frac{1}{\rho}\frac{\partial P}{\partial x} \tag{F.15}$$

$$\frac{\partial w}{\partial t} = -u\frac{\partial w}{\partial x} - w\frac{\partial w}{\partial z} \tag{F.16}$$

F.4.4 Simplifying the Equations for 1-D

At this point, we have four variables: u, w, h and P. We also have, or can get, equations for the time derivative of each (see equations F.14, F.15 and F.16; the fourth can be obtained via hydrostatic balance, which is equation F.13). We can apply the four equations to 2-D waves but to simplify things to 1-D (two variables, u and h), we'll first make the following two assumptions.

1. First, we assume that there is no vertical gradient of u (i.e., no matter how deep we go in the water, u is the same):

$$\frac{\partial u}{\partial z} = 0 \tag{F.17}$$

   This is only strictly true for very shallow waves (where the depth is much less than the wavelength). item Second, we'll allow there to be a vertical gradient of w (since it must be zero at the bottom) but we'll assume the vertical gradient is constant:

$$\frac{\partial w}{\partial z} = \frac{w_{\text{top}}}{h}$$

Using the second simplifying assumption with the continuity equation (equation F.12) gives

$$w_{\text{top}} = -h\frac{\partial u}{\partial x}$$

and plugging that into time derivative equation for h (equation F.14) gives the expression we need for h:

$$\boldsymbol{\frac{\partial h}{\partial t} = h\frac{\partial u}{\partial x} - u\frac{\partial h}{\partial x}} \tag{F.8}$$

To get the expression we need for u, we use the hydrostatic assumption (equation F.13), ignoring atmospheric pressure variations in the horizontal (meaning they are small compared to the water pressure variations),

$$\frac{\partial P}{\partial x} = \rho g \frac{\partial h}{\partial x}$$

and plug that and equation (F.17) into the time derivative equation for u (equation F.15) to get the expression we need for u:

$$\boldsymbol{\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - g\frac{\partial h}{\partial x}} \tag{F.10}$$

Problems

Problem F.1: The following code uses the forward Euler scheme applied to an object hanging from a spring (note: `koverm`, `g` and `nint` are set before these lines).

```
def accel(h):
  return -koverm*h - g
h = 0.
v = 0.
for i in range(nint):
  vnew = v + accel(h)*dt
  h = h + 0.5*(v+vnew)*dt
  v = vnew
```

When this program is run, the result is an oscillation with an amplitude that grows over time. Why does the amplitude grow over time?

Problem F.2: Suppose we wanted to modify the code shown in Problem F.1 to include drag, with an amount equal to $-bv^2$. Which line or lines would we need to modify, and why?

Problem F.3: Two students modify the code in Problem F.1 as shown below. Which method are they attempting? Do you expect there to be any difference in the results produced by the two codes? Why or why not?

```
Student A:
for i in range(nint):
  h = h + v*dt
  v = v + accel(h)*dt
```

```
Student B:
for i in range(nint):
  v = v + accel(h)*dt
  h = h + v*dt
```

Problem F.4: The integration portion of our original water wave code, which blows up, could be written as follows.

```
for i in range(N):
  dhdx[i] = ( h[(i+1)%N] - h[i-1] ) / (2 * Deltax)
  dudx[i] = ( u[(i+1)%N] - u[i-1] ) / (2 * Deltax)
for i in range(N):
  dhdt[i] = -h[i] * dudx[i] - u[i] * dhdx[i] # line 5
  dudt[i] = -u[i] * dudx[i] - g * dhdx[i] # line 6
for i in range(N):
  h[i] = h[i] + dhdt[i] * dt # line 8
  u[i] = u[i] + dudt[i] * dt
```

A student suggests we switch lines 6 and 8. Will this avoid the blow-up issue, make it worse, or have no impact? Why?

Problem F.5: Suppose we take the code in Problem F.4 (before the line switch) and add viscosity to line 5 (along with a calculation of the second derivative in the first loop). If no other changes are made (besides the addition of viscosity), will doing this avoid the blow-up issue, make it worse, or have no impact?

Index

5-point approach, 12

absorptivity, 37
albedo, 37
analytical integration, 80

backward 2-point approach, 8
Backward Euler scheme, 106
binary, 15
bit, 15

comments, 31
composite midpoint method, 89
composite trapezoidal method, 89
correlation, 69

decimal, 15
diagonalizing, 25
divided difference approach, 8
dot notation, 91

emissivity, 37

first-order, 7, 91
five-point method, 15
fixed point, 40
float, 40
floating point, 40
floating point decimal, 67
forward 2-point approach, 8
Forward Euler scheme, 105
frequency, 72

Gaussian elimination, 20, 22
gradient, 116
greenhouse effect, 36

Heun, 107
hydrostatic balance, 125

incompressible, 124
independent, 22
Infinity, 6
initial value method, 92
instruction, 2

Lagrange point, 45
leapfrog scheme, 107
loop, 3

matrix notation, 23
midpoint method, 89
midpoint value method, 93
midrange, 88
modulo operator, 115

NaN (not a number), 6
Newton's method, 61
Newton-Raphson method, 61
numerical integration, 80

oscillations, 101

partial derivative, 111
partial differential equation, 111
phase, 74

radiation balance, 37
remainder, 115
root finding, 47
roots, 47
Runge-Kutta scheme, 107

secant method, 56
second-order, 7, 92
simple harmonic motion, 101
Simpson's rule method, 95
solving by substitution, 20
solving by subtraction, 20
spring constant, 99
spring stiffness, 99
symmetric 3-point approach, 10
symmetric difference approach, 10

third, 118
three-point method, 14
time steps, 81
trapezoidal method, 89

viscosity, 118
VPython, 119

zeroes, 48