Computational Science: Getting Your Feet Wet

Jacob Jeffries

March 2022

0 Prerequisites

It's expected that you have some knowledge of calculus. This includes a solid understanding of the derivative as a limit:

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{0.1}$$

the integral as a summation (don't worry about the cumbersome notation):

$$\int_a^b f(x)\ \mathrm{d}x = \lim_{\|\Delta x\| \to 0} \sum_{i=1}^{n} f(x_i^*) \Delta x_i \tag{0.2}$$

the Taylor expansion of a function f about some point x_0:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} (x - x_0)^n \tag{0.3}$$

an understanding of what a differential equation is (not necessarily how to solve them), and the notion of a vector. The latter isn't absolutely necessary; it just really simplifies the notation. For example, if we have three functions f, g, and h that obey the differential equations $f' = f$, $g' = g$, and $h' = h$, we can simplify this down to one equation $\mathbf{r}' = \mathbf{r}$ where $\mathbf{r} = (f, g, h)$.

No programming knowledge is assumed, I will go over all the programming knowledge necessary (using Python).

1 What is computational science?

1.1 A summary

To **very** briefly summarize, computational science is using computers to solve and understand models of the natural world. This is a very popular tool, since most interesting and useful models are too complicated to solve on paper.

1.2 An example

People who have taken a physics course are familiar with the kinematic equations, which model how a projectile moves if the only force acting on the projectile is gravity:

$$\begin{aligned} x &= x_0 + u_0 t \\ y &= y_0 + v_0 t - \frac{1}{2} g t^2 \end{aligned} \tag{1.1}$$

where x and y are the horizontal and vertical positions, x_0 and y_0 are those initial positions, u_0 and v_0 are the initial horizontal and vertical velocities, g is gravitational acceleration ($9.8\,\mathrm{m/s^2}$ on Earth), and t is time. Those who especially paid attention in their physics course will know that these equations only apply under constant acceleration. Using vector notation:

$$\mathbf{r} = \mathbf{r}_0 + \mathbf{v}_0 t - \frac{1}{2}\mathbf{g} t^2 \tag{1.2}$$

where $\mathbf{r} = (x, y)$, $\mathbf{v}_0 = (u_0, v_0)$, and $\mathbf{g} = (0, g)$.

This type of simple system is what I will refer to as *analytically solvable*, i.e. one can extract the exact quantity using solely paper. What happens if we add something very realistic and important, like air resistance?

Then, we have to go back from first principles, which are Newton's laws. Note that I'll largely avoid invoking too much physics in this document, and will largely focus on solving useful mathematical models as presented as mathematical problems. The resultant differential equation that falls out of Newton's laws is:

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -k \left\| \frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} \right\| \frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} + \mathbf{g}$$

for some constant k, where $\|\cdot\|$ is the length of a vector:

$$\|\mathbf{v}\| = \sqrt{v_x^2 + v_y^2} \tag{1.3}$$

Writing out derivatives like this is cumbersome, so we'll use what Newton used for time derivatives:

$$\ddot{\mathbf{r}} = -k\|\dot{\mathbf{r}}\|\dot{\mathbf{r}} + \mathbf{g}$$

The only way to make this analytically solvable is to force the particle to only move in one dimension. Projectiles obviously don't move in one dimension, so we cannot hope to analytically solve this. I challenge the reader to try, but don't try too hard; you won't get anywhere with it.

So, what do we do? The answer is to approximate and repeat, which will be a common theme with this entire document.

2 Forward Euler Approximation

Many different systems can be modeled as first order differential equations:

$$\dot{y} = f(t, y) \tag{2.1}$$

The right hand side simply means that we can let $\dot{y}$ equal any combination of t and y - some examples:

$$\begin{aligned} \dot{y} &= y \\ \dot{y} &= t \\ \dot{y} &= y^2 + 2t \\ \dot{y} &= \sin(y) + e^{-t} \end{aligned} \tag{2.2}$$

Some might recognize the first two as analytically solvable. The others are not, though, so how do we solve them? We approximate. Note that, by definition:

$$\dot{y}(t) = \lim_{\Delta t \to 0} \frac{y(t + \Delta t) - y(t)}{\Delta t} \tag{2.3}$$

which we can approximate for some small value of Δt:

$$\dot{y}(t) \approx \frac{y(t + \Delta t) - y(t)}{\Delta t} \tag{2.4}$$

Plugging this into our general differential equation:

$$\frac{y(t + \Delta t) - y(t)}{\Delta t} \approx f(t, y(t)) \implies y(t + \Delta t) = y(t) + \Delta t f(t, y(t)) \tag{2.5}$$

Notice this is equivalent to:

$$y(t + \Delta t) \approx y(t) + \Delta t \dot{y}(t) \tag{2.6}$$

which might come off as strikingly similar to a Taylor expansion of $y(t + \Delta t)$ around t:

$$y(t + \Delta t) = y(t) + \dot{y}(t)\Delta t + \frac{1}{2}\ddot{y}(t)(\Delta t)^2 + \cdots \tag{2.7}$$

This isn't a coincidence - the more general way to come up with approximation schemes is using Taylor expansions, not the technique I used here at first. I'll save this for later, though.

So, what does this tell us? Let's say we have the value of $y(0)$. Using this, we can approximate $y(\Delta t)$:

$$y(\Delta t) = y(0) + \Delta t f(0, y(0)) \tag{2.8}$$

Similarly, we can approximate $y(2\Delta t)$:

$$y(2\Delta t) = y(\Delta t) + \Delta t f(\Delta t, y(\Delta t)) \tag{2.9}$$

and so on until we reach $N\Delta t$, which is completely controllable, meaning we can perform a very simple calculation to approximate the value of y at any value of t.

Although this calculation is very simple, it is very tedious. Especially so since you want to make Δt as small as possible, so the total number of iterations one needs to complete to reach some time t_{total} is $t_{\text{total}}/\Delta t$.

Essentially, there is a balancing act between how quickly we want the calculation to happen and how accurate we want the calculation to be. This is not a trivial task at all, and is especially difficult if you try to do the approximations by hand.

What modern tool is better at doing these tedious calculations by hand? The answer is a computer, and a computer needs to be told to do said calculations; we tell the computer how to do the calculations by writing code.

Some basic Python

2.1 Installing Python

There exists plenty of tutorials on how to install Python - I'll be avoiding going over this. I highly recommend coding in a Jupyter notebook (hint: this is what you should be looking up) to start off, it's a great way to get your feet wet.

2.2 Data types

In the same sense that we have distinct types of mathematical objects in math (e.g. a simple number versus a function, which maps numbers to other numbers), there exists data types in Python.

These types are:

- string: `str`
- integer: `int`
- float: `float`
- complex: `complex`
- list: `list`
- tuple: `tuple`
- range: `range`
- dictionary: `dict`
- set: `set`
- frozen set: `frozenset`
- bool: `bool`
- bytes: `bytes`
- byte array: `bytearray`
- memory view: `memoryview`

For our purposes, the most important of these objects will be integers, floats, lists, tuples, and ranges.

2.2.1 Integers

These are self explanatory, and are (for our purposes) the same as in math.

2.2.2 Floats

These are numbers that allow for decimals. A notable subtlety is that Python floats can only have a limited number of digits. Basically, don't try to store more than 15 decimal places in a float.

2.2.3 Lists

These are collections of any data type, and look something like `[1, 2, 3, 4]`. If you store a list, you can later directly change any element in that list.

2.2.4 Tuples

Tuples are similar to lists, but with the restriction that you can't later change an element in a tuple. Tuples look like `(1, 2, 3, 4)`.

2.2.5 Ranges

Ranges are a sequence of numbers. These allow you to go through a sequence of numbers up to some top number, which is exactly what we are doing when we do Forward Euler up to $N\Delta t$.

2.3 Creating variables

Creating variables is trivial in Python. If we want to create an integer with value 2, the line of code is:

```
x = 2
```

One can check the data type of `x`:

```
print(type(x))
```

which should print out `int`.

One notable feature is that Python will automatically detect the data type necessary to store the information you're assigning to the variable:

```
x = 2
print(type(x))
y = 2.0
print(type(y))
```

Notice that the second statement prints out `float` since we gave it decimals.

A very controversial feature of Python is that you can later change the data type of a variable:

```
x = 2
print(type(x))
x = 2.0
print(type(x))
x = [2, 2]
print(type(x))
x = (2, 2)
print(type(x))
x = range(2)
print(type(x))
```

This might seem largely innocent, but is actually one of the major reasons that Python is slower than other programming languages.

2.4 Operations

Python has built-in operations, such as addition, subtraction, multiplication, and division, which have the syntax one would expect:

```
x = 2.0
y = 1.0
sum = x + y
difference = x - y
product = x * y
```

```
6       quotient = x / y
7       print(sum, difference, product, quotient)
```

The operation with unexpected syntax is exponentiation:

```
x = 2.0
x_squared = x ** 2
x_cubed = x ** 3
sqrt_x = x ** 0.5
print(x_squared, x_cubed, sqrt_x)
```

which might take some getting used to.

3 Functions

Functions in Python hold a very similar purpose as they do in mathematics: objects that take in inputs and output some corresponding value. To define a function, use the keywords `def` and `return` as such:

```
def f(x):
   return x ** 2
```

which is equivalent to the mathematical function $f(x) = x^2$.

Functions in Python can do more general things, though, such as data type conversions:

```
def int_to_float(integer):
   integer = float(integer)
```

Where did the `return` keyword go? The answer is that the `return` keyword isn't necessary to make a function do something. Notice the difference between the two functions: `f()` is returning a value based on the input, and `int_to_float()` is directly modifying the input and changing it.

```
def f(x):
   return x ** 2

def int_to_float(integer):
   integer = float(integer)

x = 2
print(type(x))
x_squared = f(x)
print(x_squared)
int_to_float(x)
print(type(x))
x_squared = f(x)
print(x_squared)
```

Notice that, when `x` is defined, its data type is `int`, so `x_squared` is also an `int`. After applying `int_to_float()`, this subtlety changes both `x` and the value that now goes into `f()`.