DS 3: Solving Differential Equations using Python

Philip Cherian

February 3, 2020

1 Discretising a differential equation

In this Discussion Session we will attempt to describe a physical system using a computer program in Python. We will do this by discretising the differential equation, having chosen an appropriately small `dt` as discussed in the first Discussion Session. From the class, you should remember that we can approximate the derivative of a function (say, the position $x(t)$ of a particle) by:

$$\frac{\mathrm{d}x}{\mathrm{d}t} \approx \frac{\Delta x}{\Delta t},$$

which is true up to some given precision that decides how small the value of Δt is. As $\Delta t \to 0$, the approximation becomes more and more precise.

So how do we use this to solve a differential equation? Before we start, there are two important concepts to address: the *kinematics* and the *dynamics*.

1.1 Kinematics

By the kinematics of a system, we mean the equations that govern how a particle moves. These are "common-sense" equations, they arise from the very definitions of the concepts of position, velocity, and acceleration. For example, let's consider the motion of a particle in one dimension, which we assume follows some well defined function $x(t)$. We define the velocity of a particle to be the rate of change of position, therefore

$$v \equiv \frac{\mathrm{d}x}{\mathrm{d}t}.$$

When we discretise this system with a sufficiently small Δt, what we mean is that over this interval Δt, the position of the particle changes by an amount Δx, where

$$\Delta x = v(t)\Delta t.$$

Similarly, we know that the acceleration of a particle is defined by the rate of change of velocity

$$a \equiv \frac{\mathrm{d}v}{\mathrm{d}t}.$$

In our discretised scheme, this just means that the change in velocity over some small time interval Δt is given by

$$\Delta v = a(t)\Delta t,$$

and so on for all the higher derivatives of position. In both the above cases, you'll note that we use the values of the function $v(t)$ and $a(t)$ at the time t. This is because when we discretise the system, we are implicitly assuming that these functions are constant over that Δt. Furthermore, the quantities $v(t)$ and $a(t)$ might themselves change as a function of time, which would in turn change $x(t)$ and $v(t)$ respectively, and so on.

Using this technique, you can solve rudimentary problems quite simply: suppose you have a ball falling under a constant gravitational acceleration g, and let's say you don't know how to integrate the differential equation, but you would still like to know how far it fell in (say) 10 seconds. You are armed only with a pocket calculator, and so this is what you do: you draw a table like the one below, and start filling in the data. (At the time $t = 0$, let's say we start off with the ball at a height of 100 m, and we drop it from rest.)

t	0	0.1	0.2	0.3	...				
x	100				...				
v	0				...				
a	-10				...				

Table 1: The initial table, with only the first column filled in.

Now, let's say we want to populate the remaining columns, how would we go about doing this? Well, what is the change in position in the interval of time `dt` chosen? Using our analysis above, we can see that

$$\texttt{x[t+dt] = x[t] + v[t] dt} \Longrightarrow \texttt{x[0.1] = 100 + 0 x 0.1 = 100}$$

$$\texttt{v[t+dt] = v[t] + a[t] dt} \Longrightarrow \texttt{v[0.1] = 0 - 10 x 0.1 = -1}$$

Since the acceleration is constant, `a[t+dt] = a[t]`, and so we can populate the rest of the table using this technique, using only the values of velocity and acceleration from the previous time step.

t	0	0.1	0.2	0.3	...				
x	100	100	99.9	99.7	...				
v	0	-1	-2	-3	...				
a	-10	-10	-10	-10	...				

Table 2: The same table, with other columns filled in. At each time step `n dt`, the values of position and velocity are filled in, using the values of velocity and acceleration from the *previous* time step.

There are two important points to take away from this discussion: firstly, you can populate the entire table using *only* the results from the *previous* timestep; you don't have to know the entire history of the particle. And secondly, you still need the initial position and velocity of the particle. It should be easy for you to convince yourself that if you didn't start from rest, but rather threw the ball up with some initial velocity `u`, the second row of the table above would look very different. As a result, the initial position and velocity – known as the initial conditions – are essential for solving such differential equations. The fact that there are *two* such initial conditions is related to the fact that the differential equation in question is *second order*. We will deal with this in more detail in the class, but for now this should make sense in practical terms.

Of course, you might argue that this is not strictly true. Why have we forgotten about the acceleration? In this case, it was constant, but in general it could change, and without the acceleration at every time step, we wouldn't be able to define how the velocity changes! But then, you might ask, what of the rate of change of acceleration (called the "jerk")? What if this acceleration *itself* was changing as a function of time? You'd also need to know the initial "jerk", and so on and so forth. It would seem like you needed to know and infinite number of "initial" conditions – one for every derivative of the position – and it might seem an impossible task! You would be completely right. However, this is where Nature steps in to make our lives easier. It turns out that in all[1] the problems in Classical Mechanics you will be dealing with in an undergraduate physics degree, Newton's Law holds, meaning that the acceleration of an object is given by an external agent known as the *force*, F, and they are related by the mass (m) of the object through the equation:

$$a = \frac{F}{m}.$$

This defines the *dynamics* of the system, which we will speak of next.

1.2 Dynamics

Newton's Law basically states that an external agent that applies a force on an object instantaneously changes its acceleration, i.e.

$$a(t) = \frac{F(t)}{m}.$$

This means that we don't need to look for the higher derivatives of the acceleration, provided we know exactly how the force changes at any instant of time. The example above was clearly for a constant force (since the acceleration was constant). A natural question now is what woud happen if the force was different. Indeed, it is exactly this that distinguishes (say) a ball falling under gravity from a mass oscillating on a spring. This is what we call *defining the dynamics* of the problem.

Let's consider a mass and spring system. In this case, it's well known that the force is provided by Hooke's Law: $F = -kx$, where k is the spring constant, and x is the extension of the spring from its equilibrium position. This is true at any instant of time, so perhaps it's better to say that

$$F(t) = -kx(t),$$

since (in general) $x(t)$ could be a varying function, and therefore the force varies too. Using Newton's Law, this just means that the acceleration of the mass on the spring is

$$a(t) = -\frac{k}{m}x(t) = -\omega_0^2 x(t).$$

So here we have something interesting: in the previous section we showed how, given the acceleration on an object one could find the change in velocity in some time `dt`, and given the velocity of an object one could find its change in position. However, in this particular problem, we see something else: as the position of the object changes, so does the force (and consequently the acceleration)! In other words:

$$\begin{tikzcd}
\texttt{Position:} & \texttt{x[0]} \arrow[r, "\texttt{+v[0] dt}"] & \texttt{x[0.1]} \arrow[r, "\texttt{+v[0.1] dt}"] & \texttt{x[0.2]} \ldots \\
\texttt{Velocity:} & \texttt{v[0]} \arrow[r, "\texttt{+a[0] dt}"] & \texttt{v[0.1]} \arrow[r, "\texttt{+a[0.1] dt}"] & \texttt{v[0.2]} \ldots \\
\texttt{----------------} & & & \\
\texttt{Acceleration:} & -\omega_0^2\ \texttt{x[0]} \arrow[r] & -\omega_0^2\ \texttt{x[0.1]} \arrow[r] & -\omega_0^2\ \texttt{x[0.2]} \ldots
\end{tikzcd}$$

[1] Well, nearly all.

So the technique for solving such a problem would be the following:

(a) Start by defining an initial position and initial velocity (the initial conditions of the problem)

(b) Change the position of the particle, using these initial conditions.

(c) As soon as the position changes, so does the acceleration. Use this fact to update the acceleration with the new position.

(d) Use the updated acceleration to change the velocity of the object.

(e) Repeat the above process with the new position and velocity, until the total time has elapsed.

The above schematic is called the Euler(-Cromer) method. It's the simplest method by far of solving differential equations. In the class, we have also discussed the Leapfrog method, which is a lot more efficient, and a simple extension of this method. Make sure you spend some time trying to understand that. In the next section we will develop an algorithm to use the Euler method in a Python program.

> Note that in the above example we have assumed that the force is merely a function of position. In certain (important!) situations, you will see that the force (and therefore the acceleration) can also be a function of velocity. However, this is a trivial extension of what we've seen above: the acceleration will be a function of *both* `x` and `v`. I'll leave it to you as an exercise to understand how this will work.

2 Numerically solving a differential equation

> In what follows I will assume that you have read through the resources for programming so that you already have a basic understanding of what functions, NumPy arrays, and loops are, and what the syntax is to define them in Python.

The first step in solving any system is to define the dynamics, which we will do by defining an "acceleration" function in Python, which accepts some input variables, and outputs a number. For the harmonic oscillator example given above (with frequency $\omega_0 = 1$), we do this as follows:

```python
def a(pos):
    return -pos
```

This simple function accepts a variable (that we have called `pos`), and returns a number (`-1 x pos`. (I'll leave it to you to figure out how this would change if $\omega_0 \neq 1$.) Thus, if we "call" the function anywhere in the program with (say) the number "1" as an argument, we'd get a number "-1" as the output. To see if you understand it, run the following lines after defining the function above and try to explain why you get the output you do:

```python
print( a(3) )
print( a(1)*a(-1) )
print( a(17) - a(-17) )
```

```
Output: -3
        -1
        -34
```

Now that we have defined the external agent that drives our system, we can solve the kinematics. What we ultimately want to have is the data that would populate Table (2). In order to do this, we will define two arrays, one which will contain the information about the position, and one which contains the information about the velocity (we don't need one for the acceleration, since in physical systems it is always expressible as some combination of the above two quantities, as we showed earlier). We'll also make an array which contains the time elapsed, since that will make it easier for plotting later. Let's define these arrays to be NumPy arrays, since they are easier to work with for scientific purposes. You should thus import the NumPy package using the command below. (We'll also import the "pyplot" collection from the Matplotlib library, since this will help us plot things at the end.)

```
import numpy as np                # Import the NumPy package as "np"
import matplotlib.pyplot as plt   # Import the Matplotlib library
```

But how long should these arrays be? Well, that depends on two quantities, the total time `T`, and the timestep `dt`. Clearly, we are looking to discretise our system so that after some "`N`" steps we cover the time T. i.e., we require `N` to be the nearest integer that satisfies `T = N x dt`.

The arrays `t`, `x` and `v` will thus have this length, and we will initialise them to be filled with zeros, using the `np.zeros(length, data_type)` function from the NumPy library.

```
dt = 0.1                # Time step
T = 10                  # Total time over which simulation runs

N = int(T/dt)           # The function "int" takes the integer value

t = np.zeros(N,float) # The following arrays have a length of N
x = np.zeros(N,float) # and whose elements are "floats" (decimals)
v = np.zeros(N,float) # They are all initially zero.
```

We can now initialise the arrays quite simply:

```
t[0] = 0      # This is not necessary as t[0] is already zero
x[0] = 10     # Initially, the object is 10 units from the origin
v[0] = 0      # It starts from rest
```

Now that we've set up everything, we can use the definitions of position and velocity to populate the table:

```
for i in range(1, N):              # The loop doesn't touch i = 0
    t[i] = t[i-1] + dt             # Updating the time array
    x[i] = x[i-1] + v[i-1]*dt      # Updating the position array
    v[i] = v[i-1] + a(x[i])*dt     # Updating the velocity array
```

And voila, we're done! We run the loop over `N` steps, after which all the arrays are populated. At each step in the loop, the arrays are incremented by the appropriate terms, depending only on the values in the previous cell.

Notice how we invoke the dynamics (the function `a`) only when we increment the velocity: the function takes the current position, and "spits out" the current acceleration. We can now plot these arrays against `t` using the `plt.plot(x_axis_array, y_axis_array)` function:

```
plt.plot(t, x, label="Position",     color='tab:blue')
plt.plot(t, v, label="Velocity",     color = 'tab:orange')
plt.plot(t,-x, label="Acceleration",color='tab:red')
plt.xlabel("Time")
plt.legend()
plt.show()
```

Running the entire code should now produce the graph shown in Figure (1). Notice how we now have the position, velocity, and acceleration arrays as a function of time. In the last case, all we needed to know was that at every instant $a(t) = -x(t)$. (Why? How would this change if $\omega_0 \neq 1$?)

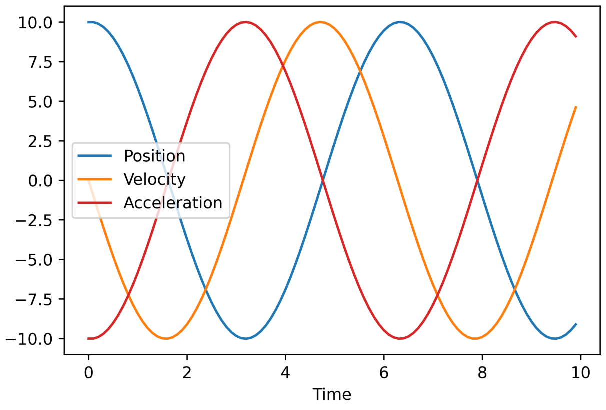

Figure 1: Position, velocity, and acceleration graphs for simple harmonic motion: notice how the acceleration graph is just (in this case) the negative of the position graph.

The entire code is given below:

```
import numpy as np                  # Import the NumPy package as "np"
import matplotlib.pyplot as plt     # Import the Matplotlib library

def a(pos):
    return -pos

dt = 0.1                 # Time step
T = 10                   # Total time over which simulation runs

N = int(T/dt)            # The function "int" takes the integer value
```

```
11
12 t = np.zeros(N,float) # The following arrays have a length of N
13 x = np.zeros(N,float) # and whose elements are "floats" (decimals)
14 v = np.zeros(N,float) # They are all initially zero.
15
16 t[0] = 0     # This is not necessary as t[0] is already zero
17 x[0] = 10    # Initially, the object is 10 units from the origin
18 v[0] = 0     # It starts from rest
19
20 for i in range(1, N):          # The loop doesn't touch i = 0
21     t[i] = t[i-1] + dt         # Updating the time array
22     x[i] = x[i-1] + v[i-1]*dt  # Updating the position array
23     v[i] = v[i-1] + a(x[i])*dt # Updating the velocity array
24
25 plt.plot(t, x, label="Position",     color='tab:blue')
26 plt.plot(t, v, label="Velocity",     color = 'tab:orange')
27 plt.plot(t,-x, label="Acceleration",color='tab:red')
28 plt.legend()
29 plt.show()
```

Let's note some important points:

(a) W have solved the problem for an object attached to a string that has initially been extended by 10 units. Note that if we wanted to solve the motion for the case when we don't stretch the spring out, but instead give the object a "kick" at $t = 0$, all we'd have to do is change the initial conditions to read `x[0] = 0`, and `v[0] = 1` (if we assume it's kicked with a velocity of +1 units).

(b) More interestingly, what if we wanted to solve the problem of a freely falling ball as we did first when we made Table (2)? What would need to change? A little bit of reflection should lead you to conclude this amazing result: *all you need to do is change line* 5! If you replace `return -pos` with `return -10`, you're solving a different problem! This comes back to what we were discussing earlier: the dynamics defines what makes this system different from any other system. The kinematics is always true, as they arise from the very *definitions* of quantities like position, velocity, and acceleration.

(c) Another useful method discussed both in class and in the Feynman Lectures on Physics, Vol. I, Sec. 9.6 (though the name is never used) is the "Leapfrog" method. You can transform the simple method described above into the much more efficient leapfrog method by simply updating the velocity by an extra half-step at the beginning. In other words, you just need to change line 18 from

```
    v[0] = 0      # It starts from rest
```

to

```
    v[0] = 0 +  a(x[0])*dt/2      # It starts from rest
```

and, remarkably, you're done!

Optional Exercise: After going through the one-body problem discussed in class, try to solve the 3-body problem in the provided Jupyter Notebook. The notebook is written in Visual Python, but much of it is incomplete; you need to fill in the appropriate commands.