Example Exercises

Gabriel S. Cabrera

August 15, 2018

Contents

1 Function Integrator

For this exercise, let us imagine we are given an arbitrary function $f(x)$, such that we wish to find its integral numerically. Let's also assume that we will be integrating between two points: `start` and `stop`, with a dx of `step`.

When using standard `Python`, we would normally integrate via the usage of a loop. We might for example choose to use the function below:

```
import numpy as np

def integrate(start, stop, step, f):
    x = start
    y = 0
    while x < stop:
        y += f(x)*step
        x += step
    return y
```

This works completely fine for smaller integrations, but what if we have an extremely large number of steps? This is when `NumPy` comes in handy – let's vectorize this process:

```
import numpy as np

def numpy_integrate(start, stop, step, f):
    x = np.arange(start, stop, step)
    return np.sum(f(x))*step
```

The output values from each integrator are slightly different, but this becomes negligeable as we increase the number of steps.

So let's test this out for a specific set of parameters – we begin with the following function:

$$f(x) = x\,(\cos(x) + 1) \tag{1}$$

Let's integrate over $f(x)$ from 0 to 100 with a step of 0.001 and see what we get[1]

```
import numpy as np

def f(x):
    return (np.cos(x) + 1)*x

def integrate(start, stop, step, f):
    x = start
    y = 0
    while x < stop:
        y += f(x)*step
        x += step
    return y

def numpy_integrate(start, stop, step, f):
    x = np.arange(start, stop, step)
    return np.sum(f(x))*step


start = 0
stop = 100
step = 0.01

I_1 = integrate(start, stop, step, f)
I_2 = numpy_integrate(start, stop, step, f)
```

Printing `I_1` gives us a result of 4948.29501615, while printing `I_2` gives us a result of 4948.29501614. The difference between these values is 1.43×10^{-9}, and therefore negligeable.

Let's now analyze how much faster our `NumPy` vectorized function is relative to the standard `Python` function by using the same problem as before, but for many different values of Δx for our `step` argument. We see in Figure 1 that increasing the number of loops improves our efficiency until it reached a maximum improvement of around 90 times!

We can therefore conclude that it is definitely worth using a `NumPy` integrator when possible, as it is always faster than using a loop.

[1] We can see a visual interpretation of the exercise in Figure 2.

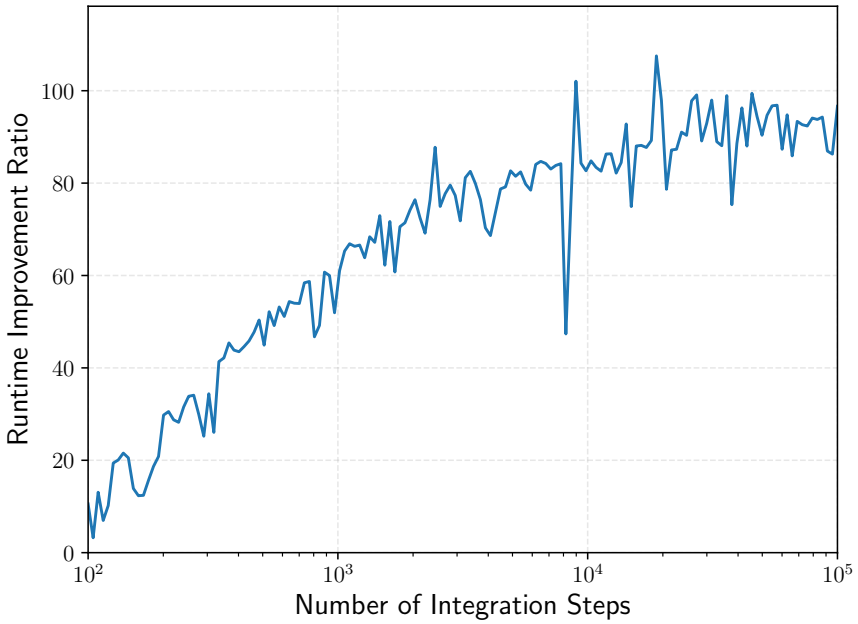

Figure 1: The factor by which our integration time improves by using `NumPy` for our integral.

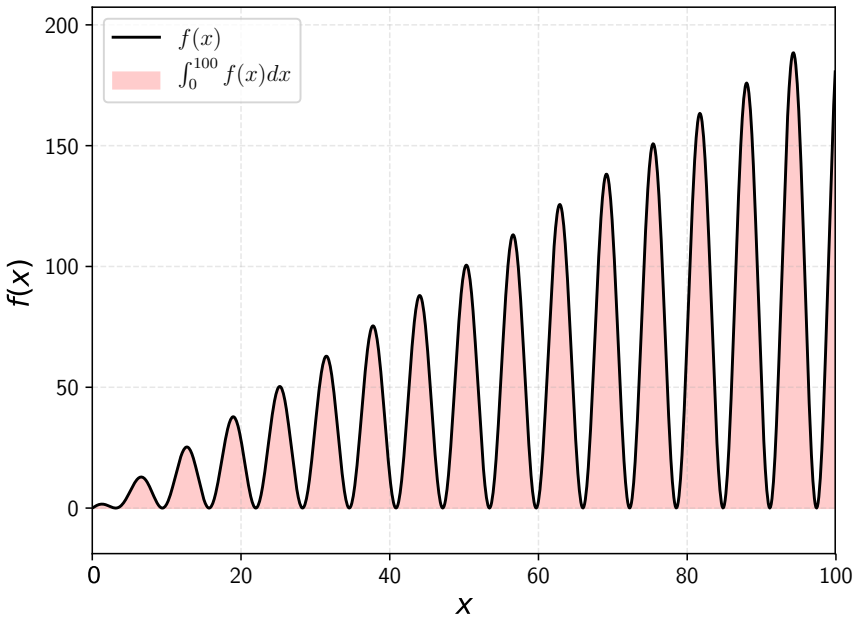

Figure 2: The integral $\int_0^{100} x\,(\cos(x)+1)\,dx$ visualized as the area under the curve for $f(x) = x\,(\cos(x)+1)$.

2 Total Charge in a Non-Uniformly Charged Cube

In beginner electrostatics, we often work with objects whose charges follow uniform distributions. We may also encounter situations where charge density distributions vary linearly or perhaps quadratically as a function of a single variable; often, these can be solved analytically through the usage of relatively simple integration methods.

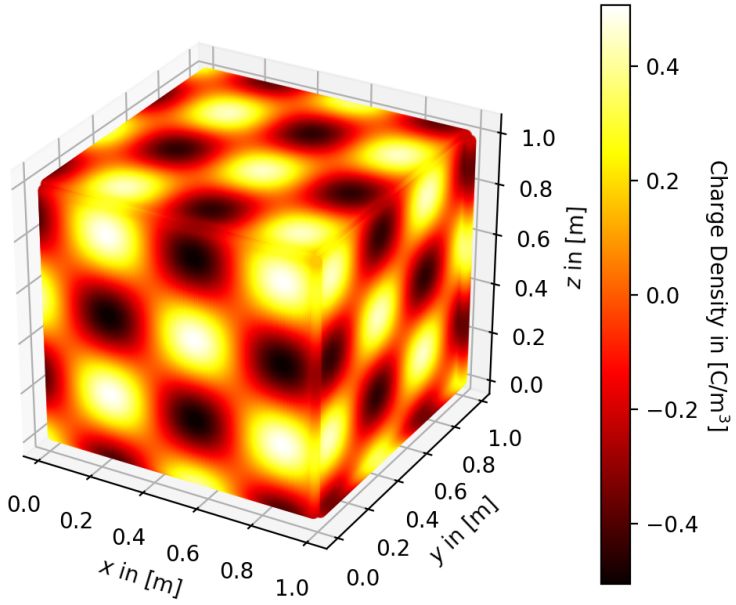

Figure 3: The charge density distribution ρ of a cube, visualized using a heatmap

Let's now assume we are given a cube whose charge density distribution varies as a function of x, y, and z – this is a case in which relying on a numerical solution may be useful. One of the reasons for this is that we can theoretically make changes to this distribution function with minimal effort, while an analytical solution will have to be reevaluated in its entirety once a small change is made to the original function. We will be taking a look at the cube shown in Figure 3. The charge distribution of this cube is governed by the following equation:

$$\rho\left(x, y, z\right) = \cos\left(10(x-10)\right)\sin\left(10(y-10)\right)\cos\left(10(z-10)\right) \tag{2}$$

This is not trivial to integrate analytically, so a numerical solution seems appropriate in this situation. To accomplish this, we need to create a `python` script; traditionally, one might create a nested loop to allow for triple-integration, but this can be avoided since we aren't dealing with a differential equation – rather, this is a perfect example of when one should apply vectorization to their script.

Before we begin, we must decide how we will be structuring our program. In this case, we

will be needing a `python`-representation of the ρ function, an array of coordinates at which we will be evaluating the charge-density, and the approximated volume differential `dV`. We will be structuring our array as follows:

$$[[x_1, y_1, z_1],\ [x_2, y_2, z_2],\ ...,\ [x_{N^3}, y_{N^3}, z_{N^3}]] \tag{3}$$

Firstly, we will need a function that will take an x, y, and z coordinate and return the density at that point. However, there is a catch; since we will be working with the above array, we need a function that can perform a vectorized evaluation of the points' charges:

```
def rho(r): #Charge density function
    x, y, z = r[:,0], r[:,1], r[:,2]
    return np.cos(10*(x-10))*np.sin(10*(y-10))*np.cos(10*(z-10))
```

Next, we need to generate this array of points; we will be using `np.linspace` and `np.meshgrid` in such a way that we end up with all combinations of all points within the confines of the cube – in other words, we need a 3D meshgrid. It is not enough however, to simply create a meshgrid – we will be needing our sets of coordinates to be structured in the way shown above. To accomplish this, we simply need to use some `NumPy` magic: a combination of the transposition command[2] and the `np.reshape` command will allow us to create a $(N^3 \times 3)$ array representing each point in our cube:

```
L = 1                                   #The cube's side length
N = 100                                 #Number of points to calculate for in each ↩
    direction

points = np.linspace(0, L, N)           #Array of all points in each direction

r = np.array(np.meshgrid(points, points, points))
r = (r.T).reshape(N**3, 3)
```

Using the given `N`, we find that our `dx` should be the length our cube divided by `N`; we will be needing `dV` however, since we are integrating over three dimensions – this can be obtained easily by finding the cube of `dx`. Finally, we will simply need to take the sum of the charge infinitessimals at each coordinate multiplied by `dV`, giving us our total charge! Here is the script in its entirety:

```
from numba import jit
import numpy as np

#INITIAL CONDITIONS

L = 1                               #The cube's side length
N = 100                             #Number of points to calculate for in each direction

def rho(r):                         #Charge density function
    x, y, z = r[:,0], r[:,1], r[:,2]
    return np.cos(10*(x-10))*np.sin(10*(y-10))*np.cos(10*(z-10))

#INTEGRATOR
points = np.linspace(0, L, N)   #Array of all points in each direction

dV = (L/N)**3                       #The volume differential

#Creating an array of all combinations of each coordinate via meshgrid/reshape
```

[2]Given an array "`a`", we can access its transpose via the command "`a.T`"

```
r = np.array(np.meshgrid(points, points, points))
r = (r.T).reshape(N**3, 3)

#Calculating the total charge in the cube
Q = np.sum(rho(r)*dV)

print('The total charge of the cube is {:g} Coulombs'.format(Q))
```

Running this program gives us the following output:

```
The total charge of the cube is 0.00237712 Coulombs
```

Did you notice that we managed to avoid any and all looping? This is the purpose of vectorization: these methods can *always* be applied as long as we are dealing with a density function that is *already known*; if we have a situation where our density function is a differential equation without a known analytical solution, it will be necessary to implement a loop[3].

3 The Half-Lives of ζ Particles

The half-life of an arbitrary system is defined as the amount of time it takes for half its elements to reach a particular state – in the case of atomic isotopes, we may wish to measure how long it takes for a nucleus to decay and release neutrons. Take Uranium-235 as an example: if we have a large quantity of this particular isotope, we may reasonably expect that half of the given atoms will decay after 703.8 million years. This means that Uranium-235 has a half life of 703.8 million years.

In this exercise we will be simulating the decay of the the fictional ζ particle – in particular, we are interested in calculating its half-life. We are given that ζ particles have a 0.1% chance of decaying per nanosecond, meaning that we can use `NumPy`'s random number generator to model their decay.

Our program will use an array (of length `N`) of ones and zeros to represent a total of N ζ particles and decayed particles, respectively.

```
N = 1e6          #Number of Particles
dt = 1           #Time-Step in nanoseconds
D = 1e-3         #Probability of Decay/dt

N = int(N)       #Converting N to integer for compatibility
p = np.ones(N)   #Array of all particles; 1 = Undecayed, 0 = Decayed
```

We will then set up a `while`-loop: it will run until the point where the sum of our array elements is less than $N/2$.

Here is an acceptable conditional:

```
while np.sum(p) > N/2:
```

For each loop iteration, we will generate an array of N random floats between 0 and 1; we will then modify this array by setting all elements smaller than 0.001 equal to zero, and all its

[3]Though using the methods outlined in Chapter **??** can significantly improve iteration speed, so take a look!

remaining elements to one – this is a way to simulate the probability of decay for each particle over the course of a nanosecond.

```
prob = random.rand(N)    #Array of Probabilities
prob[prob < D] = 0       #Sets decayed elements to zero
prob[prob != 0] = 1      #Sets all undecayed elements to one
```

Next we will multiply our particle array by our modified probability array, thus transferring the states of our particles onto the next iteration. We will also need to create a time variable `t`, which we will increase by `dt` once per iteration – this will allow us to calculate the total amount of time passed once the half-life condition has been triggered. In the end, we have our completed script:

```
from numpy import random
from numba import jit
import numpy as np

#INITIAL CONDITIONS

N = 1e6      #Number of Particles
dt = 1       #Time-Step in nanoseconds
D = 1e-3     #Probability of Decay/dt

#SIMULATION

@jit(nopython = True) #Uses Numba to improve runtime in variable-length loop
def loop(N, D, dt):

    N = int(N)       #Converting N to integer for compatibility
    p = np.ones(N)   #Array of all particles; 1 = Undecayed, 0 = Decayed
    t = 0            #Setting initial time

    #Runs the simulation until half the particles have decayed:
    while np.sum(p) > N/2:
        prob = random.rand(N)    #Array of Probabilities
        prob[prob < D] = 0       #Sets decayed elements to zero
        prob[prob != 0] = 1      #Sets all undecayed elements to one
        p = p*prob               #Multiplying decayed elements
        t += dt                  #Increases time by given time-step
    return t

#Printing our result
print('Zeta-Particles have a half-life of ~{:g} nanoseconds'.format(loop(N, D, dt)))
```

Running our program will usually[4] give us the following output:

```
Zeta-Particles have a half-life of ~692 nanoseconds
```

In conclusion, ζ particles have a half-life of 692 nanoseconds.

[4]There is a small chance it might deviate slightly from 692, since this is all a matter of probability.