University of Oslo

Scientific Programming

A Compendium

Gabriel S. Cabrera

Supervised by
Prof. Frode Hansen

August 15, 2018

Introduction

As a student, you may be exposed to difficult problems that are either not analytically solvable or simply impractical to solve by hand – this means relying on numerical methods and programming as a means to finding an approximate solution. Initially, it may appear to leave us with only one option: use a loop and run your calculations. This is not always the best way to approach a problem!

The purpose of this compendium is to give access to alternate means of running heavy calculations, either through vectorization or just-in-time compilation. Using these tools, it is possible to reduce a program's runtime significantly; this allows for extreme precision where time is of the essence, and often also leads to more elegant code.

This compendium will also review the basics behind differential equations, and introduces several numerical integration algorithms and how to use them. In short, there is a large amount of condensed information available here to those looking for ways to improve their programs' performance: both for the experienced and novice programmer.

Acknowledgements

A special thanks to Prof. F.K Hansen, J.A Fløisand, N.J Mikkelsen, and J.A Ottesen for contributions made in the creation of this text.

Contents

Chapter 1

Integration of Differential Equations

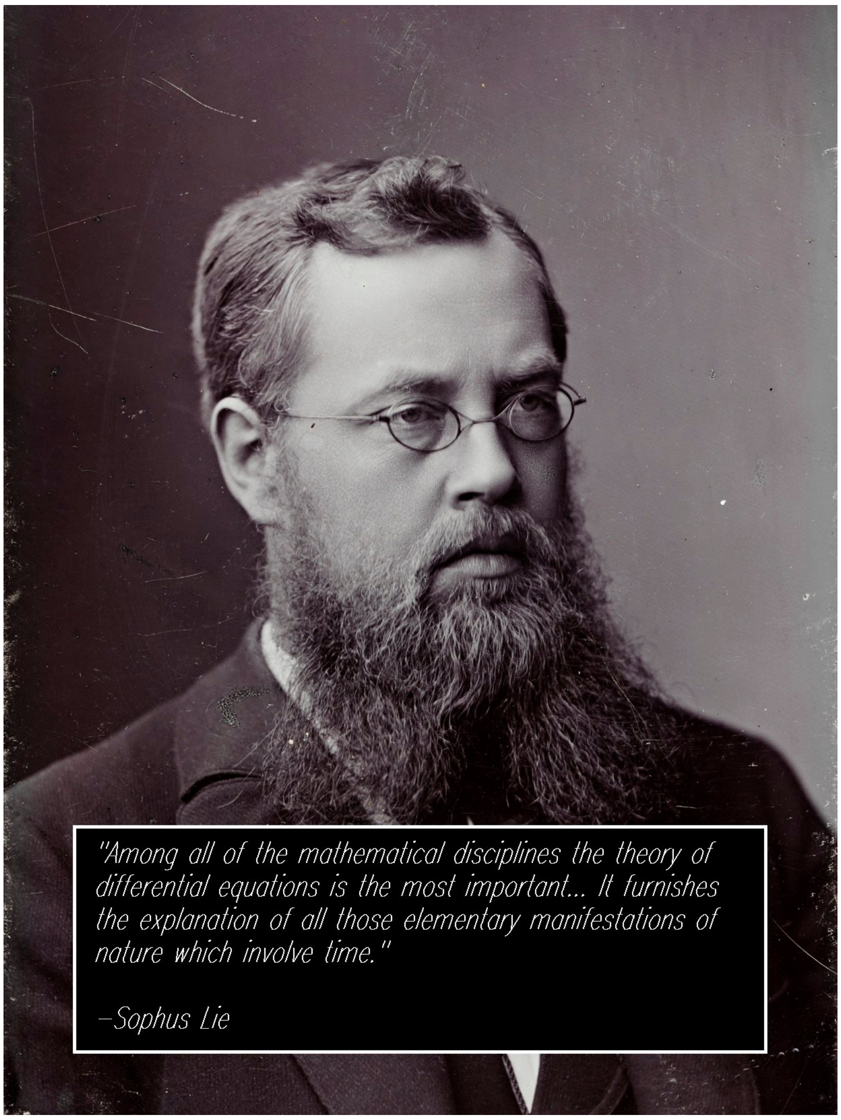
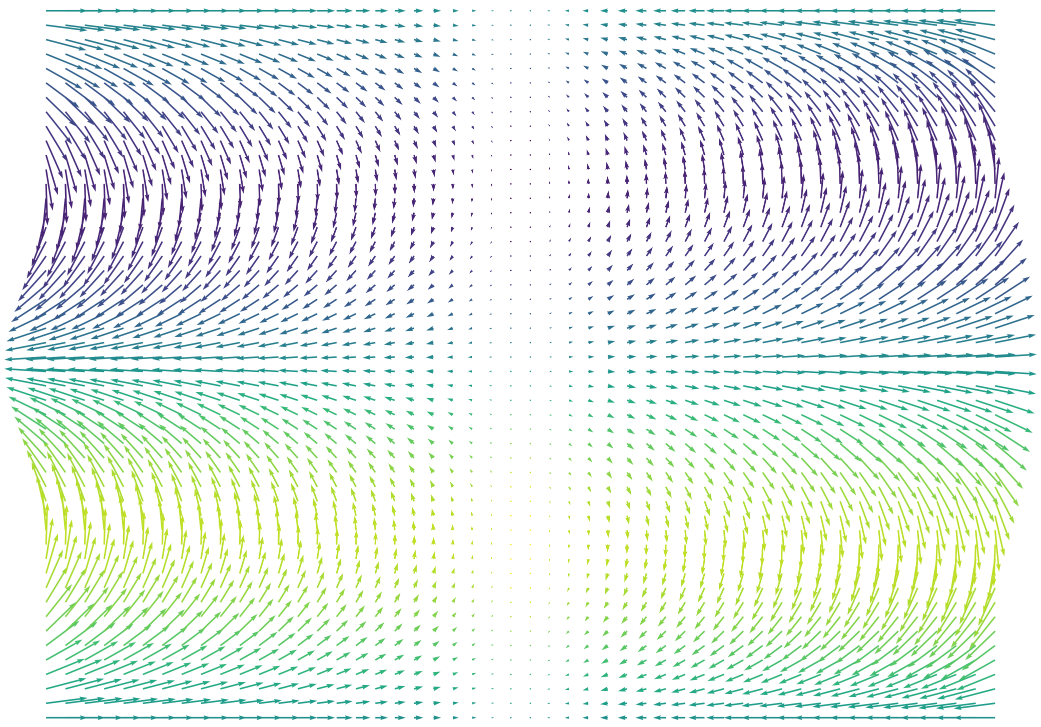

"Among all of the mathematical disciplines the theory of differential equations is the most important... It furnishes the explanation of all those elementary manifestations of nature which involve time."
–Sophus Lie

Figure 1.1: A portrait [1] of and quote by famed 19th century Norwegian mathematician Sophus Lie.

1.1 Introduction

Our universe is host to an enormous variety of physical phenomena, and although they differ in just about every way imaginable it is often possible to describe many of them using common principles.

To model many such phenomena, we must create systems composed of differential equations; these can be *ordinary* differential equations or *partial* differential equations depending on the given situation. For the sake of simplicity, we often use the abbreviations *ODE* and *PDE*, respectively.



From forces and energies, to flows, waves, and currents, we can use ODEs and PDEs to model many types of natural phenomena – in fact, any physical process with a quantity that determines its own rate of change can be modeled as a differential equation. For example, the movement of an object through a fluid is described with a differential equation – this is partially because its speed relative to the surrounding fluid determines the amount of drag it experiences. Another situation where a differential equation would be useful is when modeling an N-body system of charged particles in motion, since it would allow us to calculate the sum of the forces on each particle based on their relative distances.

All-in-all, differential equations are fundamentally engrained in nearly every branch of physics, as well as other fields such as mathematics, statistics, and many others.

1.2 Theory

Let us begin this section by going through the theory behind ODEs, since these are simpler than PDEs and are often analytically solvable; they are characterized by the fact that they contain only one independent variable – we will then briefly introduce PDEs. As an undergraduate, one is most likely to encounter a few select subcategories of ODEs, and a limited number of PDEs.

1.2.1 Linear 1st Order ODEs

This is the simplest form of an ODE, where we have a quantity and its first derivative with respect to the system's independent variable. This can be generalized as follows:

$$\frac{d}{dx}f(x) + P(x)f(x) = Q(x) \tag{1.1}$$

Where $f(x), P(x)$, and $Q(x)$ are arbitrary functions of the independent variable x.

Analytical Solution

Equations of this form are easy to work with, as they all have the following analytical solution[1]

$$f(x) = e^{-\int P(x)dx} \int e^{\int P(x)dx} Q(x)dx \tag{1.2}$$

Since these are easily solved, it can be more efficient to avoid numerical integration altogether when exposed to problems of this form, and instead calculate the analytical solution directly.

1.2.2 Linear Homogenous 2nd Order ODEs

Now, we will move on to a more tricky kind of ODE, one which contains an independent variable and its second derivative – its first derivative may also be included, depending on the purpose of the model. These can be generalized with the following:

$$\frac{d^2}{dx^2}f(x) + P(x)\frac{d}{dx}f(x) + Q(x)f(x) = 0 \tag{1.3}$$

Where $f(x), P(x)$, and $Q(x)$ are arbitrary functions of the independent variable x.

Second order linear homogenous ODEs with constant coefficients[2] can be solved analytically by finding their *characteristic polynomials*. Otherwise, it is perfectly reasonable to rely on numerical methods, which we will introduce later in this chapter.

Analytical Solution for Constant Coefficients

We are given an arbitrary linear homogenous second order ODE of the following form:

[1] Assuming, of course, that the resulting integral has a solution!

[2] This implies that $P(x)$ and $Q(x)$ are constants, rather than functions.

$$a\frac{d^2}{dx^2}f(x) + b\frac{d}{dx}f(x) + cf(x) = 0 \tag{1.4}$$

Where $f(x)$ is an arbitrary function, and $a, b, c \in \mathbb{R}$. Its characteristic polynomial is then of the following form:

$$ar^2 + br + c = 0 \tag{1.5}$$

Using our coefficients a, b, and c, we can determine the form of our solution $f(x)$. There are three distinct possibilities:

1. If $b^2 - 4ac > 0$, we have a solution with two real-valued roots r_1 and r_2, where $r_1 \neq r_2$. This gives us a solution of the following form:

   $$f(x) = Ae^{r_1 x} + Be^{r_2 x} \tag{1.6}$$

   Where A and B are determined by the initial conditions of our system.

2. If $b^2 - 4ac = 0$, we have a solution with a single root r, which is used twice in place of r_1 and r_2. Our solution in this case takes the form shown below:

   $$f(x) = Ae^{rx} + Bxe^{rx} \tag{1.7}$$

   As previously, A and B are constants for particular boundary conditions.

3. If $b^2 - 4ac < 0$, we have a solution with a pair of complex-conjugate roots r and $\bar{r}$ such that $r = \lambda + \mu i$ and $\bar{r} = \lambda - \mu i$. Now, we have a solution in another form:

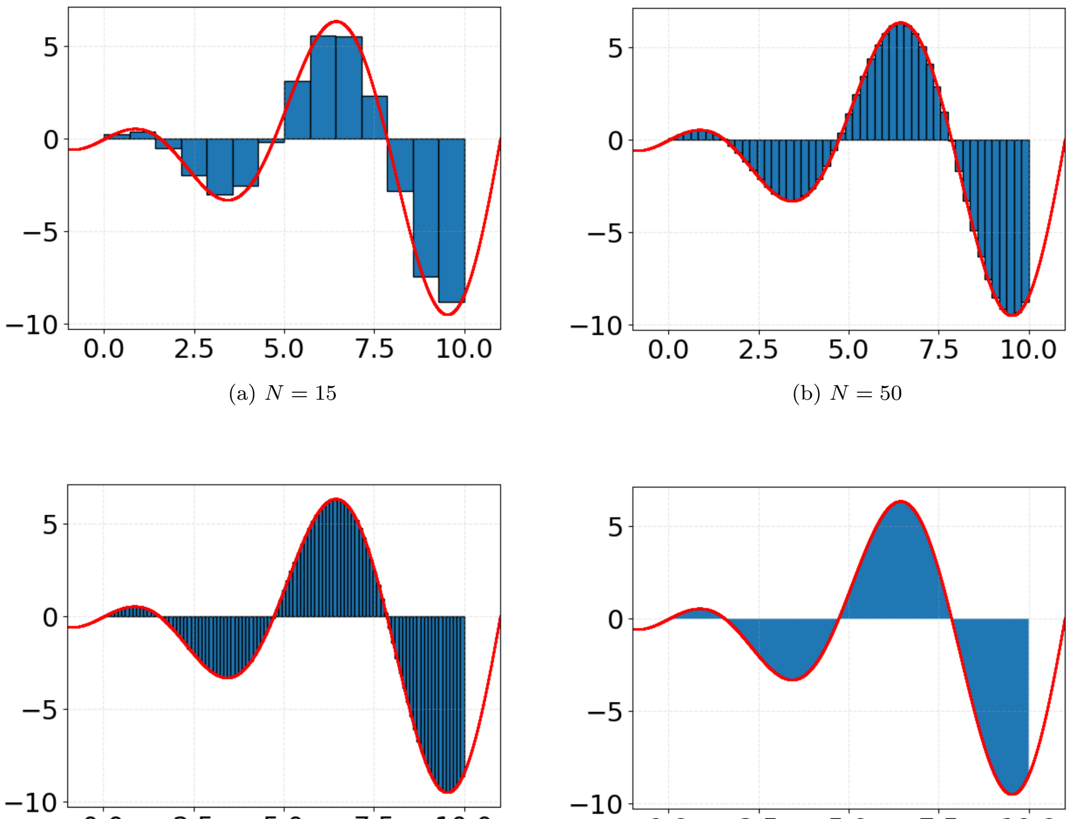
   $$f(x) = Ae^{\lambda x}\cos(\mu x) + Be^{\lambda x}\sin(\mu x) \tag{1.8}$$

   Once again, recall that A and B are found when plugging in for a set of initial conditions.

These three sets of solutions originate from the quadratic equation, where we have a square root $\sqrt{b^2 - 4ac}$; as a result, these three situations arise naturally due to the nature of the solution to the characteristic polynomial, as it is a quadratic function.

1.3 Simple Numerical Integration

Before we delve into the methods used to integrate differential equations, let us revisit the concepts behind the integration of simple one-dimensional functions. When a student is first exposed to the concept of integration, it is commonplace to use Riemann sums as a way of visualizing this process since the integral of a one-dimensional function is, in essence, the area under the function's curve. Riemann sums are an intuitive way of visualizing such ideas, and can be used to approximate integrals; we see how this works in Figure 1.2 – note that this is an example of a *midpoint* Riemann sum, since the center of each bar intercepts the plotted function, rather than its left or right corner.



(a) $N = 15$ (b) $N = 50$

(c) $N = 100$ (d) $N = 10000$

Figure 1.2: Riemann sums of varying accuracy, for $f(x) = x\cos(x)$ from 0 to 10

Given a function $f(x)$, we can use a generalized midpoint Riemann sum to find the area under its curve $F(x)$:

$$F(x) \approx \sum_{i=0}^{N} f\left(a + \frac{N}{2}\right) \Delta x \tag{1.9}$$

Where N is the number of bars in the Riemann sum, a and b are the boundaries of our integration, and $\Delta x = \frac{b-a}{N}$ is the width of each bar.

1.3.1 In Python

As we saw in (1.9), a Riemann sum is simply the process of summing over the areas of many small rectangles. Such a process is straightforward to implement in Python, as we see in the script below:

```python
def f(x):
    '''
        Insert Function Here
    '''
    return x

def integrate(f, a, b, N):
    tot = 0
    dx = (b-a)/N
    for n in range(N):
        tot += f(a + n*dx)
    return tot
```

1.4 Why Use Integration to Solve Differential Equations?

It is often rather difficult (or sometimes impossible) to solve differential equations analytically, so we must instead rely on numerical methods to find an acceptable approximation to our solution. So why use integration? Let us build an understanding by looking at a generalized 1st-order differential equation:

$$f(x(t),t) = \frac{d}{dt}x(t) \tag{1.10}$$

We are given a function $f(x(t),t)$ that is always equal to the time-derivative $\frac{d}{dt}x(t)$; turning this equation into a discrete step-wise process means we can work with our differentials more directly. Recall that the derivative is defined as follows:

$$\frac{d}{dt}x(t) = \lim_{\Delta t \to 0} \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{1.11}$$

In the case where we are working with discrete values, we may rewrite (1.10) by using the above definition (while ignoring the limit):

$$f(x(t),t) = \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{1.12}$$

We can create a discrete algorithm by solving for $x(t+\Delta t)$:

$$x(t+\Delta t) = x(t) + f(x,t)\Delta t \tag{1.13}$$

Does this seem familiar? It may, because in essence it is the approximated representation of the following integral:

$$x(t+\Delta t) = x(t) + \int_t^{t+\Delta t} f(x(t),t)dt \tag{1.14}$$

In conclusion: given an initial condition x_0, a time step Δt and a final time T, we can perform a numerical integration on a step-by-step basis with a simple algorithm, which will give us the solution to our differential equation:

Algorithm 1 A simple numerical integration algorithm

```
1: t = 0
2: while ($t \leq T$) do
3:     $x_{i+1} = x_i + f(x_i, t)\Delta t$
4:     $t_{i+1} = t_i + \Delta t$
5: end while
```

1.4.1 Programmatical Numerical Integration

We can use `python` to solve an equation of the form shown in (1.10), it is simply a matter of:

1. Creating a `python` function for $f(x(t), t)$
2. Selecting an initial condition $x(0)$
3. Finding the right balance between precision and speed, then choosing a Δt and final value T
4. Implementing Algorithm 1

Following the above steps will leave us with a script of the following form:

```python
def f(x, t):
    ...

x = ...
t = ...
dt = ...
T = ...

for i in range(int(T//dt)):
    t += dt
    x += f(x, t)*dt
```

Note that the above script is incomplete, and requires that a potential user fill in the blanks for all spots marked "`...`"; these are representative of the first three points in the aforementioned list[3].

Example 1

So how exactly can we use the above script to our advantage? Let's use the principles outlined above to solve the following differential equation:

$$\frac{d}{dt}x(t) = \cos(t) \tag{1.15}$$

Given the initial condition $x(0) = 0$, the time-step $\Delta t = 0.01$ and final time $T = 10$, we can use the following script:

[3]Those being the initial condition, time-step, and final time.

```python
import math

def f(t):
    return math.cos(t)

x = 0
t = 0
dt = 0.01
T = 10

for i in range(int(T//dt)):
    t += dt
    x += f(t)*dt

print(x)
```

This results in the following output:

```
-0.5448212197270629
```

We can compare this result to the analytical solution for (1.15):

$$x(t) = \sin(t) + C \tag{1.16}$$

Given that $x(0) = 0$, this gives us our particular solution:

$$x(t) = \sin(t) \tag{1.17}$$

We then find that $x(10) \approx -0.544$, which shows that our method was successful!

Example 2

Now that we've shown that our algorithm works as intended, let's attempt to solve a more complex differential equation:

$$\frac{d}{dt}x(t) = \sin(x+t)\cos(x-t)e^{2t} \tag{1.18}$$

Finding an analytical solution to the above is no simple task; it is preferable to rely on numerical methods. Given that $x(0) = 1$, $\Delta t = 0.001$, and $T = 5$, we have that:

```python
import math

def f(x, t):
    return math.sin(x + t)*math.cos(x - t)*math.exp(2*t)

x = 1
t = 0
dt = 0.001
T = 5

for i in range(int(T//dt)):
    t += dt
    x += f(x, t)*dt

print(x)
```

This results in the following output:

```
-431.77886890444825
```

Here we see the true beauty of numerical methods – a traditionally complex problem can be simplified significantly when given sufficient computing power!

1.5 Numerical Methods for 2nd Order ODEs

Let us now imagine that we are interested in approximating the solution to a linear, nonhomogenous second-order ODE – we can accomplish such a task by making use of integration algorithms.

All ODEs in this section will be of the following form:

$$\frac{d^2}{dt^2}f(x) + P(x)\frac{d}{dx}f(x) + Q(x)f(x) = R(x) \tag{1.19}$$

Where x is the only independent variable in our equation[4], and $f(x)$, $P(x)$, $Q(x)$, $R(x)$ are arbitrary functions of x.

For this section we will simplify our notation further by rewriting the above in a more physics-friendly manner:

$$a(t) + P(t)v(t) + Q(t)x(t) = R(t) \tag{1.20}$$

In this section, we will introduce three algorithms that can be used to numerically integrate over equations of the above form, each with its own strengths and weaknesses. This section will be mostly theoretical, such that in-depth examples will be brought up in a subsequent section.

1.5.1 Euler-Cromer

One of the simplest methods available to us is Euler-Cromer integration; if you are looking for a fast and simple algorithm, Euler-Cromer is well suited to the task. On the other hand, it has a lack of precision when compared to other algorithms, and is not a good fit for systems where the conservation of energy is of vital importance.

- **Speed**: `Excellent`
- **Simplicity**: `Excellent`
- **Precision**: `Bad`
- **Energy**: `Not Conserved`

[4]Differential equations of multiple independent variables are PDEs, as introduced in **??**, and so involve more complexity.

To use this algorithm, we must start with a set of initial conditions x_0 and v_0. We also need to choose a time-step[5] Δt, and a total runtime T[6].

Algorithm 2 The Euler-Cromer integration algorithm for a linear nonhomogenous second-order ODE

```
1: t = 0
2: while ($t \leq T$) do
3:     $a_{i+1} = R(t_i) - P(t_i)v_i - Q(t_i)x_i$
4:     $v_{i+1} = v_i + a_{i+1}\Delta t$
5:     $x_{i+1} = x_i + v_{i+1}\Delta t$
6:     $t_{i+1} = t + \Delta t$
7:     $i = i + 1$
8: end while
```

In `python`, we can set up an integration loop as a function:

```python
def integrator(x0, v0, T, dt):

    def P(t):
        ...

    def Q(t):
        ...

    def R(t):
        ...

    N = int(T//dt)

    t = dt
    x = x0
    v = v0

    for i in range(N):
        a = R(t) - P(t)*v - Q(t)*x
        v += a*dt
        x += v*dt
        t += dt

    return x
```

To use the above function, be sure to set up functions[7] for `P(t)`, `Q(t)`, and `R(t)`.

1.5.2 Runge-Kutta 4

In situations where we are in need of more precise results on a step-wise basis[8], we can use the fourth order Runge-Kutta algorithm[9]. As with the Euler-Cromer algorithm, RK4 is not energy-conserving – in addition to this, it doesn't run as quickly as Euler-Cromer due to its complexity.

[5] A smaller time-step will lead to more accurate results, but will increase a program's runtime.
[6] We can assume that our initial time is $t = 0$.
[7] These can be replaced with constants if necessary.
[8] In other words, if we wish to increase accuracy without reducing Δt.
[9] We will use the abbreviation RK4.

RK4 is very useful in situations such as when solving partial differential equations, or finding the solutions for more complex systems that require extra precision.

- **Speed**: `Bad`
- **Simplicity**: `Bad`
- **Precision**: `Excellent`
- **Energy**: `Not Conserved`

Algorithm 3 The fourth order Runge-Kutta integration algorithm for a linear nonhomogenous second-order ODE

```
1: t = 0
2: while ($t \leq T$) do
3:     $x_\alpha = x_i$
4:     $v_\alpha = v_i$
5:     $a_\alpha = R(t_i) - Q(t_i)x_\alpha - P(t_i)v_\alpha$
6:
7:     $x_\beta = x_\alpha + v_\alpha \frac{\Delta t}{2}$
8:     $v_\beta = v_\alpha + a_\alpha \frac{\Delta t}{2}$
9:     $a_\beta = R(t_i + \frac{\Delta t}{2}) - Q(t_i + \frac{\Delta t}{2})x_\beta - P(t_i + \frac{\Delta t}{2})v_\beta$
10:
11:    $x_\gamma = x_\alpha + v_\beta \frac{\Delta t}{2}$
12:    $v_\gamma = v_\alpha + a_\beta \frac{\Delta t}{2}$
13:    $a_\gamma = R(t_i + \frac{\Delta t}{2}) - Q(t_i + \frac{\Delta t}{2})x_\gamma - P(t_i + \frac{\Delta t}{2})v_\gamma$
14:
15:    $x_\delta = x_\alpha + v_\gamma \Delta t$
16:    $v_\delta = v_\alpha + a_\gamma \Delta t$
17:    $a_\delta = R(t_i + \Delta t) - Q(t_i + \Delta t)x_\delta - P(t_i + \Delta t)v_\delta$
18:
19:    $a_{\text{avg}} = \frac{1}{6}(a_\alpha + 2a_\beta + 2a_\gamma + a_\delta)$
20:    $v_{\text{avg}} = \frac{1}{6}(v_\alpha + 2v_\beta + 2v_\gamma + v_\delta)$
21:
22:    $v_{i+1} = v_\alpha + a_{\text{avg}}\Delta t$
23:    $x_{i+1} = x_\alpha + v_{\text{avg}}\Delta t$
24:
25:    $t_{i+1} = t + \Delta t$
26:    $i = i + 1$
27: end while
```

There are other ways to describe 4^{th}-order Runge-Kutta integration, but the above shows each step of the process in depth rather than combining each step to maximize legibility. If one prefers to perform each step inline, that is also acceptable.

RK4 can be implemented in Python in a variety of ways; the skeleton code below is one of many methods that can be used, following the exact step-by-step method shown in the above algorithm:

```python
def integrator(x0, v0, T, dt):

    def P(t):
        ...

    def Q(t):
        ...

    def R(t):
        ...

    N = int(T//dt)

    t = dt
    x_1 = x0
    v_1 = v0
    dt_2 = dt/2

    for i in range(N):
        a_1 = R(t) - Q(t)*x_1 - P(t)*v_1

        t += dt_2
        x_2 = x_1 + v_1*dt_2
        v_2 = v_1 + a_1*dt_2
        a_2 = R(t) - Q(t)*x_2 - P(t)*v_2

        x_3 = x_1 + v_2*dt_2
        v_3 = v_1 + a_2*dt_2
        a_3 = R(t) - Q(t)*x_3 - P(t)*v_3

        t += dt_2
        x_4 = x_1 + v_3*dt
        v_4 = v_1 + a_3*dt
        a_4 = R(t) - Q(t)*x_4 - P(t)*v_4

        a_avg = (a_1 + 2*a_2 + 2*a_3 + a_4)/6
        v_avg = (v_1 + 2*v_2 + 2*v_3 + v_4)/6

        v_1 = v_1 + a_avg*dt
        x_1 = x_1 + v_avg*dt

    return x_1
```

Once again, the above script is incomplete and requires the user to create functions for `P(t)`, `Q(t)`, and `R(t)`.

1.5.3 Leapfrog

So far we've explored two integration algorithms – unfortunately, both of them are unable to model the conservation of energy; to remedy this, we can use **Leapfrog** integration[10]. Relative to Euler-Cromer and RK4, this algorithm is a middle-ground of sorts with regards to its complexity, efficiency, and precision – it is its ability to conserve energy that makes it invaluable, such as in the simulation of planetary orbits or the motion of an oscillating spring.

- **Speed**: `Moderate`
- **Simplicity**: `Moderate`
- **Precision**: `Moderate`
- **Energy**: `Conserved`

[10]Leapfrog integration is actually a type of Verlet integration, more information available here: https://en.wikipedia.org/wiki/Verlet_integration

Algorithm 4 The Leapfrog integration algorithm for a linear nonhomogenous second-order ODE

```
1: t = 0
2: while ($t \leq T$) do
3:     $a_{i+1} = R(t_i) - P(t_i)v_i - Q(t_i)x_i$
4:     $x_{i+1} = x_i + v_i\Delta t + \frac{1}{2}a_i\Delta t^2$
5:     $v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1})\Delta t$
6:     $t_{i+1} = t + \Delta t$
7:     $i = i + 1$
8: end while
```

To model this in `python`, we can use the script below:

```python
def integrator(x0, v0, T, dt):

    def P(t):
        ...

    def Q(t):
        ...

    def R(t):
        ...

    N = int(T//dt)

    t = dt
    x = x0
    v = v0
    a_i = R(t) - P(t)*v - Q(t)*x

    for i in range(N):
        t += dt
        x += v*dt + 0.5*a_i*dt**2
        a_iplus1 = R(t) - P(t)*v - Q(t)*x
        v += 0.5*(a_i + a_iplus1)*dt
        a_i = a_iplus1

    return x
```

As in all our previous examples, be sure to create functions for `P(t)`, `Q(t)`, `R(t)`.

Chapter 2

NumPy

2.1 Motivation

This section can be considered somewhat independently of the rest of this chapter; for the experienced but rusty, it is intended as a quick-reference. For the inexperienced it can be seen as a motivator – it highlights many of the key concepts one should understand about vectorization, and is a showcase of what a novice can accomplish once they've read through this chapter.

2.1.1 Basic Arithmetic

Addition, subtraction, multiplication, division, and exponentiation is possible between two equal-sized arrays *or* between an array and a `float/int`:

Input

```python
import numpy as np

a = np.array([2, 3, 6, 8])
b = np.array([8, 6, 2, 4])

print(a + b)
print(a + 1)

print(b - a)
print(a - 1)

print(a * b)
print(a * 2)

print(a / b)
print(a / 2)

print(a ** b)
print(a ** 2)
```

Output

```
[10  9  8 12]
[3 4 7 9]
[ 6  3 -4 -4]
[1 2 5 7]
[16 18 12 32]
[ 4  6 12 16]
[0.25 0.5  3.   2.  ]
[1.  1.5 3.  4. ]
[ 256  729   36 4096]
[ 4  9 36 64]
```

2.1.2 Miscellaneous Operations

Numpy has countless other available operators, such as the square root (`sqrt`), natural exponential function (`exp`), the natural logarithm (`log`), and the base-10 logarithm (`log10`).

Input

```python
import numpy as np

a = np.array([1, 4, 9, 16])

print(np.sqrt(a))
print(np.exp(a))
print(np.log(a))
print(np.log10(a))
```

Output

```
[1. 2. 3. 4.]
[2.71828183e+00 5.45981500e+01 8.10308393e+03 8.88611052e+06]
[0.         1.38629436 2.19722458 2.77258872]
[0.         0.60205999 0.95424251 1.20411998]
```

2.2 Introduction

When processing data, one must choose between a straightforward, simple, but slower language (such as `Python` or `Lua`) or a more complex but faster language (such as `C++` or `Fortran`) – simplicity (usually) results in less efficient usage of a computer's resources.

Fortunately there is a middle ground for `Python`: the `NumPy` module. Generally speaking, working with sets of numbers in `Python` requires us to iterate through lists, a slow process relative to looping in lower-level languages such as `C++`, but `NumPy` allows us to *vectorize* our lists such that they can be processed as single units allowing for more elegant code and much faster processing.

There is a downside to `NumPy` however: vectorized arrays are static in length – once created, an array's length is unchangeable. This means that we lose both the ability to append new elements to an array, as well as the ability to delete existing ones. In addition to this, `NumPy` is unable to improve performance for the integration of second-order differential equations[1].

[1] To improve performance in such cases, take a look at the `Numba` module.

Example Let's say we have two sets of numbers of equal length – if we wish to add them to each other element-wise, `Python` normally requires us to loop through them as follows:

```python
a = [0,1,2,3,4,5,6,7,8,9]
b = [9,8,7,6,5,4,3,2,1,0]

c = []
for i,j in zip(a, b):
    c.append(i+j)
```

To perform the same operation as above in `NumPy` is a lot simpler, as shown below:

```python
import numpy as np

a = np.array([0,1,2,3,4,5,6,7,8,9])
b = np.array([9,8,7,6,5,4,3,2,1,0])

c = a + b
```

In this case, `NumPy` is slightly slower than standard `Python`[2], but we quickly see that it is advantageous as we make the lists longer and longer – in fact, Figure 2.1 shows that `NumPy` is on average 12 times more efficient than using a `Python` loop when using arrays of sufficient length. This efficiency only gets better as we deal with more complex operations!

[2]Though only by a fraction of a second

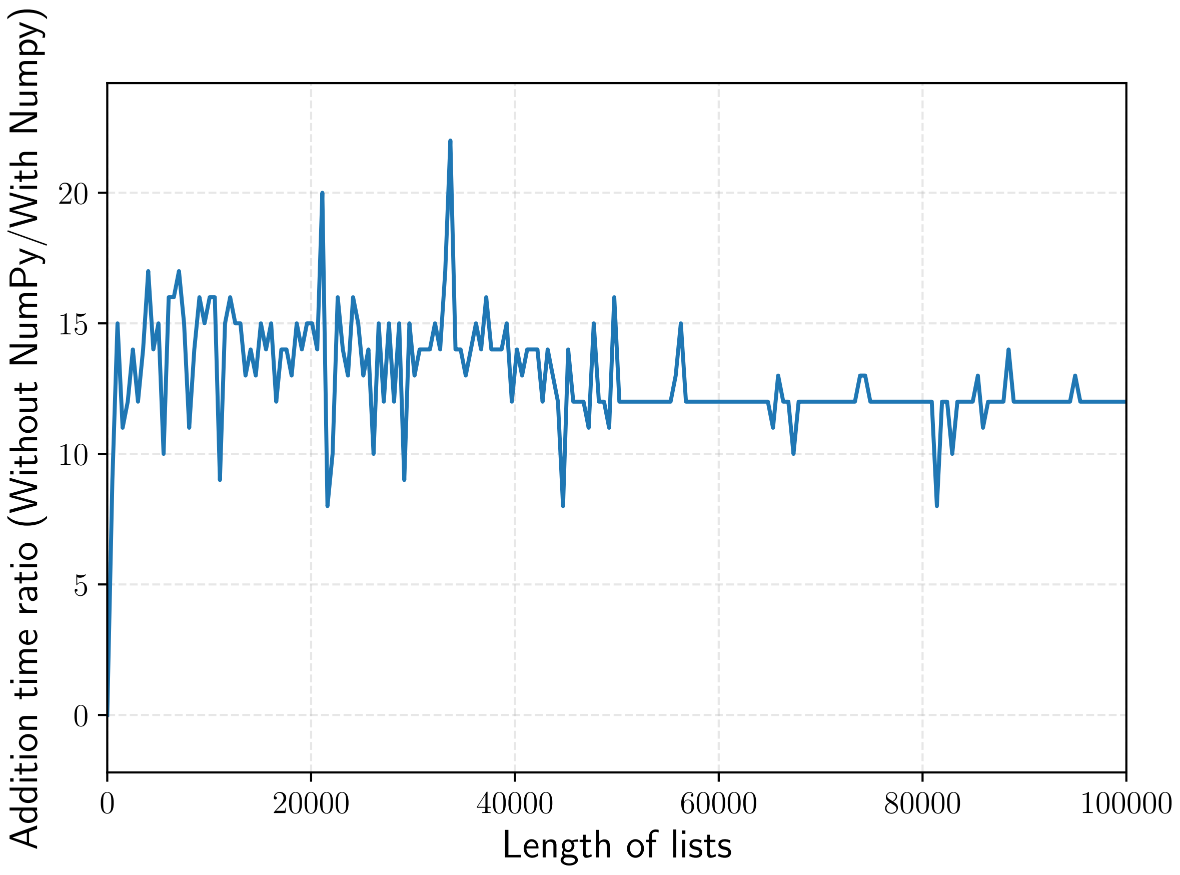


Figure 2.1: How `NumPy` can improve performance in array addition based on the lengths of given arrays

2.3 Converting Lists to NumPy Arrays

In `NumPy`, it is possible to generate arrays of all shapes, sizes, and dimensions. Everything from vectors to matrices to tensors can be recreated as arrays, however we should start by understanding 1-D arrays before moving on to higher dimensions.

2.3.1 One Dimension

To understand 1-D `array` objects, we will begin by defining a generalized vector:

Let $\vec{v} \in \mathbb{R}^n$ for $n \in \mathbb{N}$; this means that we have a vector $\vec{v}$ of length n, where n is any whole-number greater than zero. Our vector can also be represented by the following notation:

$$\vec{v} = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} \tag{2.1}$$

In `NumPy`, 1-D arrays are *row-vectors* as opposed to *column-vectors*; this will become relevant later when we look at 2-D arrays.

Example Vectors can take many forms, such as $\vec{a} = \begin{bmatrix}1 & -2 & 5\end{bmatrix}$ or $\vec{b} = \begin{bmatrix}0.01 & 0.5\end{bmatrix}$ or $\vec{c} = \begin{bmatrix}5 & 2.5 & 81 & 32 & -33.5\end{bmatrix}$.

We can recreate these vectors in `array` form, as shown below:

```python
import numpy as np

a = np.array([1, -2, 5])
b = np.array([0.01, 0.5])
c = np.array([5, 2.5, 81, 32, -33.5])
```

2.3.2 Two Dimensions

As with `lists` and `tuples`, we can create 2-D arrays with `NumPy` – alternatively, we can also create `matrix`-objects[3], though this will not be our focus.

Once again, let us take a look at a natural mathematical analogy: matrices. Let's describe them in three different ways:

Formally Let $A \in \mathbb{R}^{m\times n}$ be an $m \times n$ matrix of elements $a_{i,j} \in \mathbb{R}$ with $m, n \in \mathbb{N}$, $m \geq i \in \mathbb{N}$, and $n \geq j \in \mathbb{N}$.

Intuitively We have a matrix A consisting of real numbers with m rows and n columns.

Visually

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \tag{2.2}$$

In `NumPy`, 2-D arrays are created in much the same way as 1-D arrays:

Example 1 Let's say we have the 3×3 identity matrix:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

We can use `NumPy` to create a corresponding 2-D array, we must simply separate I_3 row-by-row:

```python
import numpy as np

I_3 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
```

[3]For an overview of the differences between `array` and `matrix` objects, see: https://www.scipy.org/scipylib/faq.html#what-is-the-difference-between-matrices-and-arrays

Where printing `I_3` gives us the following output:

```
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Example 2 We are given the following 6×4 matrix:

$$\begin{bmatrix} 3 & 3 & 1 & 6 \\ 7 & 4 & 0 & -3 \\ 0 & -2 & 9 & 7 \\ -2 & 3 & 0 & 1 \\ 5 & 7 & 1 & 1 \\ 8 & 8 & 1 & -8 \end{bmatrix} \tag{2.4}$$

We can once again separate our matrix into row-vectors to create a 2-D array:

```python
import numpy as np

M = np.array([[3, 3, 1, 6], [7, 4, 0, -3], [0, -2, 9, 7], [-2, 3, 0, 1], [5, 7, 1, ↩
    1], [8, 8, 1, -8]])
```

Printing `M` gives us the output below:

```
[[ 3  3  1  6]
 [ 7  4  0 -3]
 [ 0 -2  9  7]
 [-2  3  0  1]
 [ 5  7  1  1]
 [ 8  8  1 -8]]
```

2.3.3 Column Vectors

Recall from Section 2.3.1 that 1-D `NumPy` arrays are row-vectors; this means that if we wish to create a column vector we must set up a 2-D array with single-element arrays.

Opting to use column vectors by default may come in handy when programming for linear algebra[4], since it may reduce the amount of effort required to perform operations between vectors and matrices.

Example We are given the unit vectors for $\mathbb{R}^3$ as follows:

$$\hat{x} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \hat{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \hat{z} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.5}$$

To model these vectors using `NumPy`, we must create a set of 2-D arrays each consisting of 3 nested arrays, each of which only contain a single element. A functioning script is shown below:

[4] If you are intending to program with a heavy focus on linear algebra, it may be worthwhile to look into `NumPy`'s `matrix` object.

```python
import numpy as np

x = np.array([[1], [0], [0]])
y = np.array([[0], [1], [0]])
z = np.array([[0], [0], [1]])
```

Printing `x`, `y`, and `z` gives us the following output:

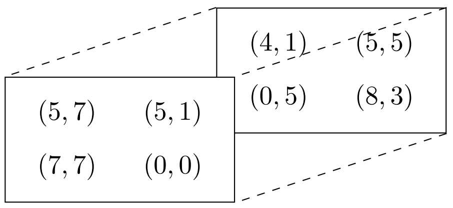
```
[[1]
 [0]
 [0]]

[[0]
 [1]
 [0]]

[[0]
 [0]
 [1]]
```

2.3.4 Higher Dimensions

We've now described the process used to generate 1-D and 2-D arrays, so what's next? Specifically, `NumPy` limits us to a maximum[5] dimension of 32, though this should not be an issue in most situations.

There are many uses for multidimensional arrays such as tensor[6] operations, dealing with partial differential equations, or modeling a system of objects in a field. Let us look at a few examples:

Example 1 Let's model a 4th-order multidimensional array, an object that can be visualized as a 4-D matrix. Here we have an example of a $2 \times 2 \times 2 \times 2$ array:



To create this using `NumPy` we must simply nest our lists in the desired hierarchy, making sure to use the row-vectors once more:

```python
import numpy as np

M = np.array([[[[5, 7], [5, 1]], [[7, 7], [0, 0]]],
[[[4, 1], [5, 5]], [[0, 5], [8, 3]]]])
```

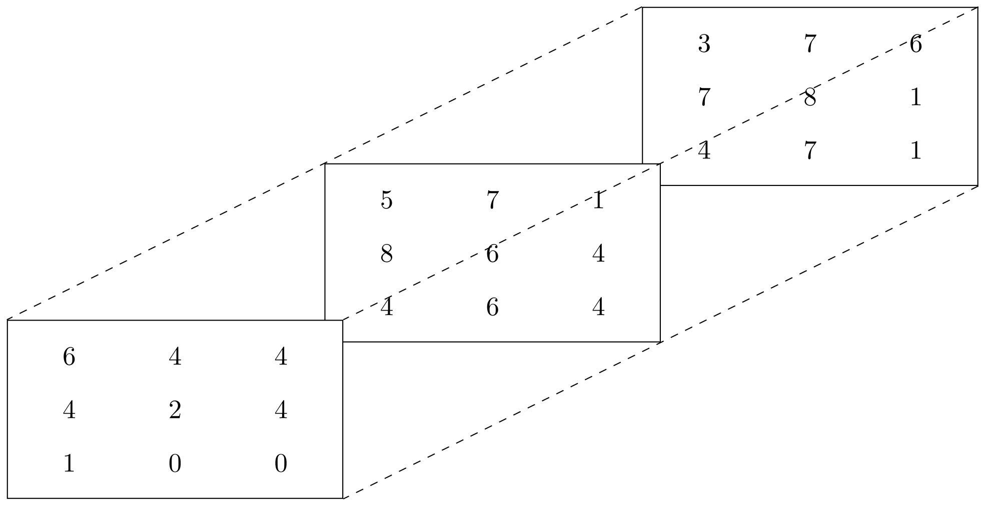
Printing the `M` array gives us the following output:

[5] Attempting to initialize an array with more than 32 dimensions will raise an exception – `ValueError: sequence too large; must be smaller than 32`

[6] Information on Tensors available at Wikipedia: https://en.wikipedia.org/wiki/Tensor

```
[[[[5 7]
   [5 1]]

  [[7 7]
   [0 0]]]


 [[[4 1]
   [5 5]]

  [[0 5]
   [8 3]]]]
```

Example 3 Let's now create a 3rd-order multidimensional array, which is similar to a 3-D matrix. Let us take a look at a $3 \times 3 \times 3$ array:



To create this using `NumPy`, we must once again nest our lists, while being careful to maintain the correct dimensions for our array:

```python
import numpy as np

M = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4], [4, 6, 4]], ↩
    [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])
```

Printing the `M` array gives us the following output:

```
[[[6 4 4]
  [4 2 4]
  [1 0 0]]

 [[5 7 1]
  [8 6 4]
  [4 6 4]]

 [[3 7 6]
  [7 8 1]
  [4 7 1]]]
```

2.4 Array Properties and Usage

2.4.1 Array Shape

In `NumPy` there is a standardized system that is used to describe an `array`'s dimensions.
In mathematical notation, we would say that the following is a 3×2 matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \tag{2.6}$$

This can be reproduced in `NumPy` via the following:

```python
import numpy as np

M = np.array([[1, 2], [3, 4], [5, 6]])
```

Printing `M` gives us the output below:

```
[[1 2]
 [3 4]
 [5 6]]
```

In `NumPy`, we would say that the dimensions of the `array` equivalent to Equation 2.6 is `(3,2)`; this is called the array's *shape*[7], which can be found by accessing the array's `shape` attribute:

```python
import numpy as np

M = np.array([[1, 2], [3, 4], [5, 6]])
print(M.shape)
```

This gives us the following output:

```
(3, 2)
```

So in short, the array shape tells us the "depth" of our object in each of its directions, with the order of these depths shown in the array's `shape` tuple. Programmatically speaking, the order of these values is determined by the nesting order chosen during the array's initialization.

2.4.2 Getting Elements

Extracting and modifying individual elements from a `NumPy` array is relatively straightforward, as the required syntax is equivalent to the syntax used for lists and tuples, but it is also important to know how to extract or modify entire columns, rows, or planes in an array. In fact, we can extract any homogenous[8] sub-array from `NumPy` arrays.

[7]More information on the `shape` attribute can be found on `SciPy`'s website: `https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html`

[8]Homogeneity in the sense that all its rows are of equal size in their respective depths.

Let's begin with reviewing how to get individual elements. Here are a few quick examples as a reminder:

1-D Arrays

Let's take a look at the following vector:

$$\begin{bmatrix} 4 & 2 & 7 & 5 & 3 & 1 \end{bmatrix} \tag{2.7}$$

Printing 5:

```python
import numpy as np

a = np.array([4, 2, 7, 5, 3, 1])
print(a[3])
```

2-D Arrays

We will be examining the following matrix:

$$\begin{bmatrix} 5 & 3 & 8 \\ 4 & 3 & 5 \\ 3 & 6 & 1 \end{bmatrix} \tag{2.8}$$

Single Elements Printing 6:

```python
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
print(A[2][1])
```

Subarrays Printing `[5 3 8]`:

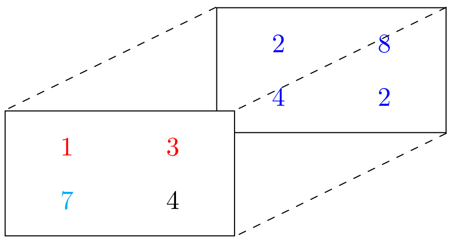
```python
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
print(A[0])
```

For an $m \times n$ array `A`, the following guide will give us an element's index:

$$\begin{bmatrix} \texttt{A[0][0]} & \texttt{A[0][1]} & \cdots & \texttt{A[0][n-1]} \\ \texttt{A[1][0]} & \texttt{A[1][1]} & \cdots & \texttt{A[1][n-1]} \\ \vdots & \vdots & \ddots & \vdots \\ \texttt{A[m-1][0]} & \texttt{A[m-1][1]} & \cdots & \texttt{A[m-1][n-1]} \end{bmatrix} \tag{2.9}$$

3-D Arrays

We will be working with the following 3-D matrix:



Single Elements Printing 7:

```python
import numpy as np

A = np.array([[[1, 3], [7, 4]], [[2, 8], [4, 2]]])
print(A[0][1][0])
```

1-D Subarrays Printing `[1 3]`:

```python
import numpy as np

A = np.array([[[1, 3], [7, 4]], [[2, 8], [4, 2]]])
print(A[0][0])
```

2-D Subarrays Printing `[[2 8] [4 2]]`:

```python
import numpy as np

A = np.array([[[1, 3], [7, 4]], [[2, 8], [4, 2]]])
print(A[1])
```

Higher Dimensional Arrays

The principles introduced for 1-D, 2-D, and 3-D arrays will carry over to higher dimensions, the trick is to really know the shape of your array before implementing your code. In other words, printing your array's `shape` (as introduced in Section 2.4.1) can be a useful aid in determining exactly where your desired values are located.

2.4.3 Getting Sub-Arrays

Getting an array's elements is relatively straightforward, but let's say we wish to extract the second column of a matrix – this sort of task can be accomplished inefficiently and in a convoluted manner by creating an empty array and appending it with the second element of each row while looping through it, but we are interested in efficiency and elegance.

NumPy gives us such an option via the use of slice objects, whereby a user can define a range of values to extract from an array. Slice objects in this case are defined as follows: [start:stop:step], where each of these values must be of type int. In addition to this, we have that:

- If the slice is defined as [start:stop] the step is set equal to 1 by default.
- It is possible to leave either start or stop or *both* blank; in such a case, start would be assigned a default value of 0, while stop would include all subsequent elements past start.
- We can use the step option on an entire array with [::step]. Using a slice such as [::2] would reference an array's every other element.
- The step can be negative! This simply reverses the order of iteration through the array and then applies the chosen step – be sure to make start $>$ stop however.
- Using a slice such as [::-step] will reverse the entire array, and iterate through it with the given step.
- For multidimensional arrays we can use the same syntax, but with a comma dividing individual "blocks". For example, a 2-D array could have a slice formatted in such a way that [start1:stop1:step1:, start2:stop2:step2], where the depth of the array being referenced increases with each added block.

As previously, this syntax is best taught via example, so we will show how to extract values in 1-D, 2-D, and 3-D arrays.

1-D Arrays

We are given the following vector:

$$\begin{bmatrix} 4 & 2 & 7 & 5 & 3 & 1 \end{bmatrix} \tag{2.10}$$

Printing [2 7 5]:

```python
import numpy as np

a = np.array([4, 2, 7, 5, 3, 1])
print(a[1:4])
```

Note Take a close look at our subarray a[1:4], and note that the first boundary integer 1 is the same as the index of 2 within our original array a. Now, observe that our second boundary integer 4 is greater than the index of 5 within our original array a.

In slicing syntax, the first boundary integer should therefore correspond to the lower index of our sub-array, while the second boundary integer should be the higher index of the sub-array +1.

2-D Arrays

We have the following matrix:

$$\begin{bmatrix} 5 & 3 & 8 \\ 4 & 3 & 5 \\ 3 & 6 & 1 \end{bmatrix} \tag{2.11}$$

Columns Printing `[3 3 6]`:

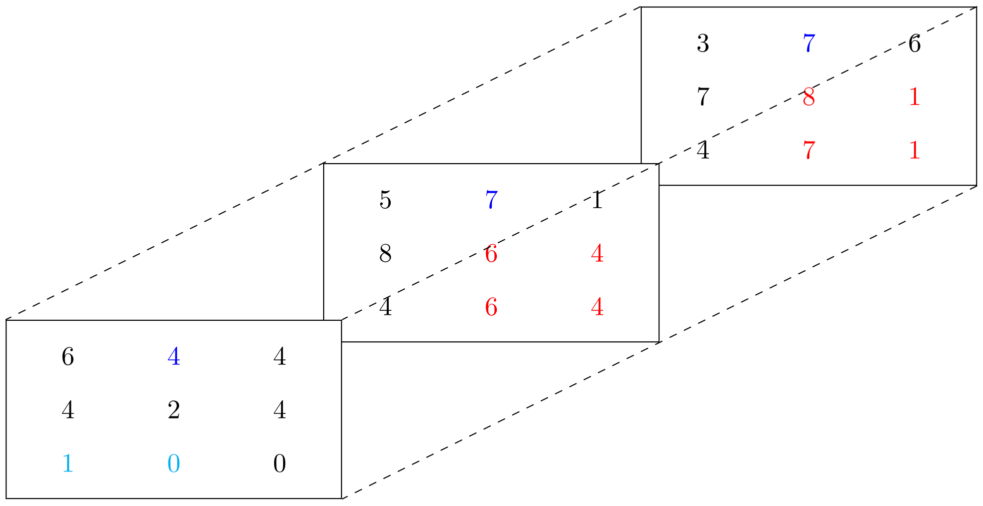
```
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
print(A[:,1])
```

Subarrays Printing `[8 5]`:

```
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
print(A[0:2,2])
```

3-D Arrays

Things start to get complex when dealing with 3-D arrays, since each added dimension vastly increases the slicing possibilities[9]. We will be working with the following 3-D matrix:



Rows Printing `[4 7 7]`:

[9]Things gets even more complex as we hit 4-D arrays, since these are far more difficult to visualize.

```python
import numpy as np

M = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4],
[4, 6, 4]], [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])

print(M[:,0,1])
```

Cubes Printing `[[[6 4] [6 4]] [[8 1] [7 1]]]`:

```python
import numpy as np

M = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4],
[4, 6, 4]], [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])

print(M[1:,1:,1:])
```

More Rows Printing `[1 0]`:

```python
import numpy as np

M = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4],
[4, 6, 4]], [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])

print(M[0,2,:-1])
```

2.4.4 Setting Elements and Sub-Arrays

This section will be brief, as it builds upon Sections 2.4.2 and 2.4.3.

Since arrays in `NumPy` have a fixed size upon initialization, we must rely on changing elements directly rather than deletion and appending with `del` and `append`, respectively. In the aforementioned sections, we described how to access elements and sub-arrays via the usage of indices and `slice` objects – the theory here is the same, and only requires a slight change in syntax.

In short, we define an index/slice "`i`" of an array "`a`" and set it to a value "`v`" whose shape must be broadcastable[10] to that of the selected slice[11] with the syntax `a[i] = v`.

We will use a few examples to illustrate this more clearly.

1-D Arrays

Example 1 We are given the following array:

$$\begin{bmatrix} 4 & 2 & 7 & 5 & 3 & 1 \end{bmatrix} \tag{2.12}$$

We can change 2 into 0 with the following:

[10] Broadcasting is discussed later in Section 2.4.6. For now, think of a broadcastable shape as one that matches the shape of the other, but in any desired orientation.

[11] Of course, if we are replacing an individual number, then our value should also be a number, not an array.

```python
import numpy as np

a = np.array([4, 2, 7, 5, 3, 1])
a[1] = 0
```

Printing `a` gives us the output below:

```
[4 0 7 5 3 1]
```

Example 2 Continuing from the previous example, we now have the following array:

$$\begin{bmatrix} 4 & 0 & 7 & 5 & 3 & 1 \end{bmatrix} \tag{2.13}$$

Let's say we wish to replace 7, 5, and 3 with 1, 3, and 5. This can be accomplished by first creating an array matching the three new values: `[1 3 5]`

```python
import numpy as np

a = np.array([4, 0, 7, 5, 3, 1])
b = np.array([1, 3, 5])
a[2:5] = b
```

Printing `a` gives us the output below:

```
[4 0 1 3 5 1]
```

2-D Arrays

We are given a matrix:

$$\begin{bmatrix} 5 & 3 & 8 \\ 4 & 3 & 5 \\ 3 & 6 & 1 \end{bmatrix} \tag{2.14}$$

We can swap out the column containing `[3 3 6]` for another array[12] `[1 2 3]` with the following script.

```python
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
b = np.array([1, 2, 3])
A[:,1] = b
```

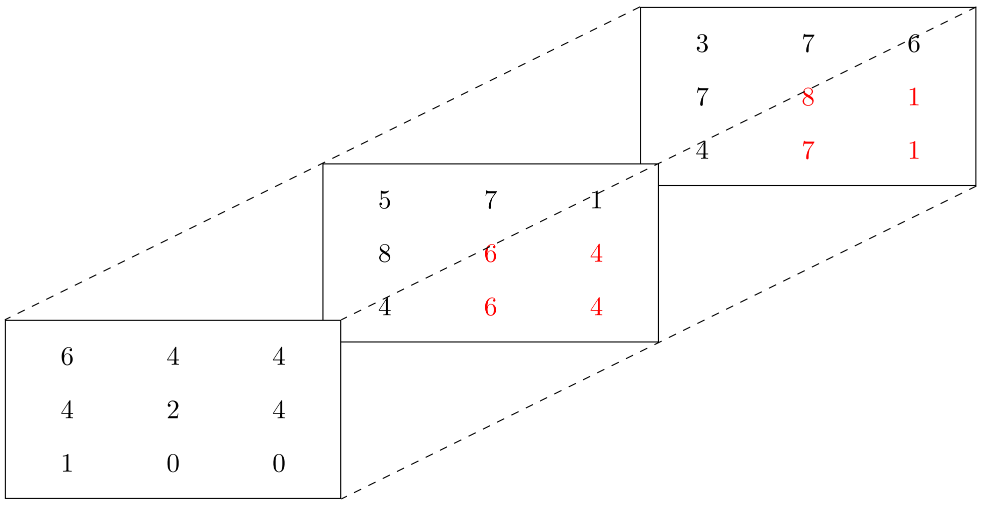
Printing `A` then gives us the following output:

[12]Even though we are technically swapping a column for a row, this still works due to the concepts we will bring up in Section 2.4.6.

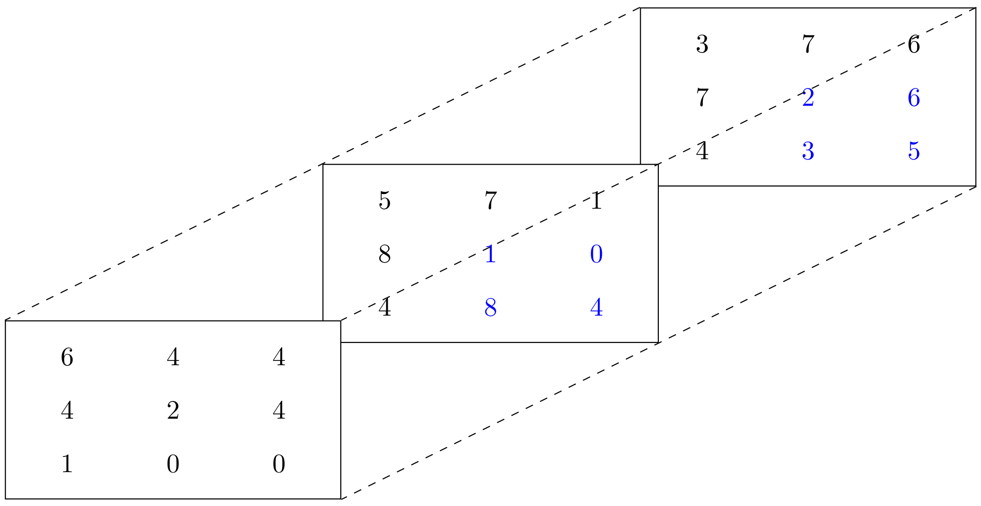
```
[[5 1 8]
 [4 2 5]
 [3 3 1]]
```

3-D Arrays

We are given a 3-D matrix:



We are interested in switching out our matrix' back right corner, creating a new matrix below:



To make these changes efficiently, we must create a $2 \times 2 \times 2$ array and set up the `slice` for the red elements' positions to this new array: `[1:,1:,1:]`.

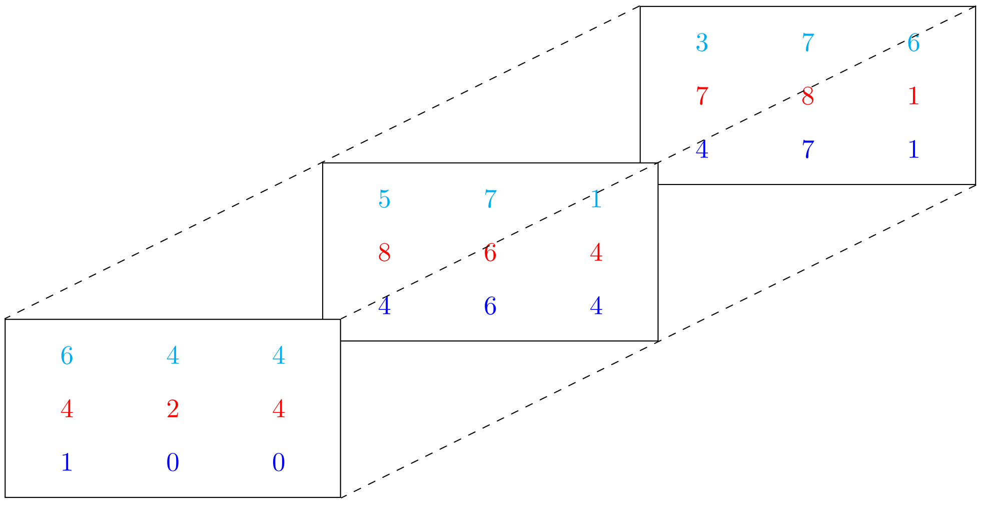
We have a script to accomplish this for us:

```
import numpy as np

A = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4], [4, 6, 4]], ↩
    [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])

b = np.array([[[1, 0], [8, 4]], [[2, 6], [3, 5]]])

A[1:,1:,1:] = b
```

2.4.5 Axis

When dealing with multidimensional arrays, there are situations where a `NumPy` function needs to know in which direction it should operate. This is where the array `axis` comes in – the concept is rather simple once we understand how `shape` works, since the `axis` of an array is simply the term defining whether we are referring to a row, column, or other direction when dealing with 3+ dimensions. In short, the axis is simply the index given for an array direction as defined by the order in its `shape` attribute.

Example We are given the following 3-D array:



Each color represents one of the elements grouped over our array's 1st axis, this is because our first axis is perpendicular to the planes we've color coded, thus defining the selected axis. Orthogonality can always be used to determine which axis we are working with, though this can be difficult to picture in higher dimensions.

In 3-D, we can picture that an operation will occur between corresponding members of each `axis` in the plane defined by the `axis` direction. When dealing with N dimensions, the operation will occur between all parallel (N-1) dimensional sub-arrays orthogonal to the given `axis`.

We can take the sum over the color coded 1st axis with the script below:

```
import numpy as np

```

```python
A = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4], [4, 6, 4]], ↩
    [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])
print(np.sum(A, axis = 1))
```

This will add each colored group to each other, giving us the following output:

```
[[11  6  8]
 [17 19  9]
 [14 22  8]]
```

2.4.6 Broadcasting

In Section 2.4.4, we were able to swap out a column in a matrix with a simple row vector. But how is this possible if the arrays have different shapes? It's because NumPy uses a clever compatibility system called **broadcasting** that allows operations to be performed between arrays of similar yet different shapes or between scalars and arrays. This is why we can multiply scalars by matrices in NumPy, because the scalar's value is *broadcasted* to the shape of the array, allowing for element-wise multiplication to occur.

It is highly recommended that new users read the documentation provided by SciPy on the matter: https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html

2.4.7 Datatypes

To improve a program's runtime, NumPy requires that array elements have a static type – by default, an array will try and determine what type it should contain upon initialization, when it detects the types in its input list/tuple. We can however force an array to take on a specific type, a process that is often overlooked at critical moments.

Example Let's say we wish to create a linearly spaced array with the linspace function referenced in Section 2.7.8, then redefine an element by adding a complex scalar to it.

```python
import numpy as np

a = np.linspace(1, 10, 10)
a[5] = a[5] + 1j
print(a)
```

Attempting to run the above script will result in a warning and an undesirable output, as shown below:

```
script.py:4: ComplexWarning: Casting complex values to real discards the imaginary part
  a[5] = a[5] + 1j
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

If we wish to be sure our array can handle complex variables, we need to set its dtype[13] to complex upon initialization. So to fix the above script, we implement the following:

[13] A shortening of *datatype*

```python
import numpy as np

a = np.linspace(1, 10, 10, dtype = complex)
a[5] = a[5] + 1j
print(a)
```

And now we get the output we desire:

```
[ 1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+1.j  7.+0.j  8.+0.j  9.+0.j
 10.+0.j]
```

The lesson to learn here is that we must always take care to create arrays that can handle any datatypes that we intend to insert into them as elements.

2.5 Logic

When dealing with simple objects in `Python` such as `bool`s, `float`s, or `int`s, we often take for granted the simplicity with which we can use comparators such as `==`, or `>=`. This is something that takes getting used to in `NumPy` (and when using arrays in general), since we now have different ways in which to compare arrays.

2.5.1 Element-Wise Comparators

The comparators in this subsection are used by calling the desired function directly from **NumPy** and passing two equally shaped arrays as arguments. Its output will then be a boolean array of equal shape, with each pair of elements evaluated in their respective indices.

Example We are given two arrays `a` and `b`; to check which of their elements are equal element-wise, we can use the `equal` function:

```python
import numpy as np

a = np.array([[1, 2, 3],[4, 5, 6]])
b = np.array([[1, 2, 2],[4, 5, 5]])
c = np.equal(a, b)
```

Printing `c` gives us the following output:

```
[[ True  True False]
 [ True  True False]]
```

Table of Comparators Since we don't have the luxury of using all our standard comparators, we can use Table 2.1 as a guide. We can refer to it when we wish to make element-wise comparisons between two arrays `a` and `b`.

Table 2.1: Element-wise comparators for two equal-shaped `NumPy` arrays `a` and `b`

Standard `Python`	`NumPy` Arrays
`a == b`	`equal(a, b)`
`a != b`	`not_equal(a, b)`
`a > b`	`greater(a, b)`
`a < b`	`less(a, b)`
`a >= b`	`greater_equal(a, b)`
`a <= b`	`less_equal(a, b)`

2.5.2 Element-Wise Boolean Operators

If we intend to set up more complex logical statements, we will need to have access to logical operations. Let us once again begin with an example:

Example Let's negate[14] an array `a` with the `logical_not` function.

```python
import numpy as np

a = np.array([[True, True], [False, False]])
b = np.logical_not(a)
```

Printing `b` gives us the output below:

```
[[False False]
 [ True  True]]
```

Table of Operators This section can be referred to when looking for an element-wise logical operation.

Table 2.2: Element-wise logical operations for two equal-shaped `NumPy` arrays `a` and `b`

Standard `Python`	`NumPy` Arrays
`not a`	`logical_not(a)`
`a and b`	`logical_and(a, b)`
`a or b`	`logical_or(a, b)`
`a ^ b`	`logical_xor(a, b)`

2.5.3 Other Comparators

There are a few other comparators that may come in handy:

[14]Meaning, we will use the element-wise equivalent to the `not` operation.

Table 2.3: Element-wise logical operations for two equal-shaped `NumPy` arrays `a` and `b`

`NumPy` Syntax	Description
`array_equal(a, b)`	`True` if `a` and `b` have the same shape and elements.
`array_equiv(a, b)`	`True` if `a` and `b` are broadcastable with the same elements.

2.6 Arithmetic

`NumPy`'s ability to vectorize arithmetic is arguably its most useful feature – it allows us to run operations such as addition, subtraction, multiplication, division, and exponentiation between any two broadcastable arrays in a single step. We can also perform other operations on individual arrays, such as taking their square root with `sqrt`, or setting it up as an exponent in `exp`.

Table of Operations Here is an overview of operations that can be taken between two broadcastable arrays `a` and `b`.

Table 2.4: Arithmetic operations for two broadcastable `NumPy` arrays `a` and `b`

Operation	Simple `NumPy` Syntax	`NumPy` Function
Addition	`a + b`	`np.add(a, b)`
Subtraction	`a - b`	`np.subtract(a, b)`
Multiplication	`a * b`	`np.multiply(a, b)`
Division	`a / b`	`np.divide(a, b)`
Floor Division	`a // b`	`np.floor_divide(a, b)`
Modulus	`a % b`	`np.mod(a, b)`
Exponentiation	`a ** b`	`np.power(a, b)`

Other useful mathematical functions can be found in `SciPy`'s documentation: https://docs.scipy.org/doc/numpy-1.14.0/reference/routines.math.html

2.7 Useful Functions

Here is an overview of some useful functions a `NumPy` user should be familiar with; it is recommended that one read the descriptions for any unfamiliar ones, as they may come unexpectedly handy.

2.7.1 absolute

Takes the absolute value of an array element-wise – in cases with complex elements, it will find their real-valued absolute value.

So given the following:

```python
import numpy as np

a = np.array([1, -3, 5, 3+4j, -8, 5+12j])
print(np.absolute(a))
```

We get this resulting output:

```
[ 1.   3.   5.   5.   8.  13.]
```

2.7.2 amax

Given an array and an axis, finds the element-wise maximum values.

Let's say we wish to find the maxima for `axis = 2`, then we can use the following:

```python
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.amax(a, axis = 2))
```

Giving us our output:

```
[[2 4]
 [6 8]]
```

2.7.3 amin

Given an array and an axis, finds the element-wise minimum values.

Let's say we wish to find the minima for `axis = 2`, then we can use the following:

```python
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.amin(a, axis = 2))
```

Giving us our output:

```
[[1 3]
 [5 7]]
```

2.7.4 arange

Creates an array of linearly-spaced values from `x0` up to (but not necessarily including[15]) `x1`, with a step of `dx` between them.

To create an array of values between -25 and 25 with a step of 1.5, we can use the following:

```python
import numpy as np

x0 = -25
x1 = 25
```

[15] `x1` may not always be included in the resulting array, since `x1` may not be evenly divisible by `dx`.

```python
dx = 1.5

a = np.arange(x0, x1, dx)
```

Printing `a` give us the following output:

```
[-25.   -23.5 -22.   -20.5 -19.   -17.5 -16.   -14.5 -13.   -11.5 -10.    -8.5
  -7.    -5.5  -4.    -2.5  -1.     0.5   2.     3.5   5.     6.5   8.     9.5
  11.    12.5  14.    15.5  17.    18.5  20.    21.5  23.    24.5]
```

As in the case with `linspace`, we are able to use non-integer steps in `arange` generated arrays.

2.7.5 argmax

Similar to `amax`, with one crucial difference – rather than directly returning the maxima, returns the indices for the maxima.

Let's say we wish to find the maximum indices for `axis = 2`, then we can use the following:

```python
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.argmax(a, axis = 2))
```

Giving us our output:

```
[[1 1]
 [1 1]]
```

2.7.6 argmin

Similar to `amin`, with one crucial difference – rather than directly returning the minima, returns the indices for the minima.

Let's say we wish to find the minimum indices for `axis = 2`, then we can use the following:

```python
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.argmin(a, axis = 2))
```

Giving us our output:

```
[[0 0]
 [0 0]]
```

2.7.7 diff

Given an array, takes the difference between each sub-array and its preceeding sub-array for a given `axis`. This is best explained visually, so let's take a look at the following 2-D matrix:

$$\begin{bmatrix} 5 & 3 & 8 \\ 4 & 3 & 5 \\ 3 & 6 & 1 \end{bmatrix} \tag{2.15}$$

Let's say we wish to find the difference between each colored column, how can we accomplish this? Using Section 2.4.5 as a guide, we must apply our function on the axis orthogonal to the groups we wish to operate on – in this case, we see that the columns are separated along the 0th axis, implying that we should use `diff` on the 1st axis:

```python
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
print(np.diff(A, axis = 1))
```

This gives us the following output:

```
[[-2  5]
 [-1  2]
 [ 3 -5]]
```

We can also infer by the signs of each element above that `diff` performs its operation such that we subtract the elements at lower indices from the elements in higher indices, in this case subtracting from right to left.

2.7.8 linspace

Creates an array of `N` linearly-spaced values between `x0` and `x1`.

To create an array of 101 values from -25 to 25, we can use the following:

```python
import numpy as np

x0 = -25
x1 = 25
N = 101

a = np.linspace(x0, x1, N)
```

Printing `a` give us the following output:

```
[-25.   -24.5 -24.   -23.5 -23.   -22.5 -22.   -21.5 -21.   -20.5 -20.   -19.5
 -19.   -18.5 -18.   -17.5 -17.   -16.5 -16.   -15.5 -15.   -14.5 -14.   -13.5
 -13.   -12.5 -12.   -11.5 -11.   -10.5 -10.    -9.5  -9.    -8.5  -8.    -7.5
  -7.    -6.5  -6.    -5.5  -5.    -4.5  -4.    -3.5  -3.    -2.5  -2.    -1.5
  -1.    -0.5   0.     0.5   1.     1.5   2.     2.5   3.     3.5   4.     4.5
   5.     5.5   6.     6.5   7.     7.5   8.     8.5   9.     9.5  10.    10.5
  11.    11.5  12.    12.5  13.    13.5  14.    14.5  15.    15.5  16.    16.5
  17.    17.5  18.    18.5  19.    19.5  20.    20.5  21.    21.5  22.    22.5
  23.    23.5  24.    24.5  25.  ]
```

The advantage in using `linspace` is that we can have non-integer steps between each array element, something that cannot be accomplished directly via the `range` command.

2.7.9 maximum

Given two arrays of equal shape, finds their element-wise maximum values.

Here is an example:

```python
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(np.maximum(a, b))
```

Giving us our output:

```
[[5 6]
 [7 8]]
```

2.7.10 meshgrid

Allows the user to quickly create two mesh arrays from two linearly-spaced 1-D arrays. As an example, let's create two arrays using `linspace` and pass them to meshgrid and see what is created:

```python
import numpy as np

a = np.linspace(0, 5, 6)
b = np.linspace(0, 3, 4)
X, Y = np.meshgrid(a, b)
```

Printing `X` give us the following result:

```
[[0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]]
```

While `Y` give us a similar array to `X`, only in terms of the y-direction:

```
[[0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1.]
 [2. 2. 2. 2. 2. 2.]
 [3. 3. 3. 3. 3. 3.]]
```

Meshgrids come in handy when dealing with functions of multiple variables, and works for any number of dimensions, just be sure to have enough variables ready for the resulting output[16].

2.7.11 minimum

Given two arrays of equal shape, finds their element-wise minimum values.

[16]For instance, we would have needed an `X`, `Y`, and `Z` variable if we'd chosen to pass three linspaces as meshgrid arguments.

Here is an example:

```python
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(np.minimum(a, b))
```

Giving us our output:

```
[[1 2]
 [3 4]]
```

2.7.12 ones

Creates an array of ones in the desired shape (in the form of an integer or array of integers).

To create a 2×2 array of ones, we can use the script below:

```python
import numpy as np

a = np.ones((2, 2))
```

Printing our array leaves us with our desired output:

```
[[1. 1.]
 [1. 1.]]
```

2.7.13 ones_like

Creates an array of ones in the shape of an input array.

To create a 2×2 array of ones with `ones_like`, we can use the script below:

```python
import numpy as np

a = np.ones_like([[1, 2], [3, 4]])
```

Printing our array leaves us with this output:

```
[[1. 1.]
 [1. 1.]]
```

2.7.14 prod

Takes the product of an array for a given `axis` – if the `axis` is not defined, it will take the total product over all its elements.

If we have a simple 1-D array, using `prod` is straightforward; this process gets complex quickly as we increase our array dimension, so it is recommended that the user be familiar with Section 2.4.5.

For a 3-D array, we can take the product over its axis `0` with the following:

```python
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.prod(a, axis = 0))
```

Since its zero-th axis refers to the set of sub-arrays highest up in the hierarchy, we end up with the products of `[[1, 2], [3, 4]]` and `[[5, 6], [7, 8]]`, giving us the following output:

```
[[ 5 12]
 [21 32]]
```

2.7.15 sum

Takes the sum of an array for a given `axis` – if the `axis` is not defined, it will take the total sum over all its elements.

If we have a simple 1-D array, using `sum` is straightforward; this process gets complex quickly as we increase our array dimension, so it is recommended that the user be familiar with Section 2.4.5.

For a 3-D array, we can take the sum over its axis `0` with the following:

```python
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.sum(a, axis = 0))
```

Since its zero-th axis refers to the set of sub-arrays highest up in the hierarchy, we end up with the sums of `[[1, 2], [3, 4]]` and `[[5, 6], [7, 8]]`, giving us the following output:

```
[[ 6  8]
 [10 12]]
```

2.7.16 zeros

Creates an array of zeros in the desired shape (in the form of an integer or array of integers).

To create a 2×2 array of zeros, we can use the script below:

```python
import numpy as np

a = np.zeros((2, 2))
```

Printing our array leaves us with our desired output:

```
[[0. 0.]
 [0. 0.]]
```

2.7.17 zeros_like

Creates an array of zeros in the shape of an input array.

To create a 2×2 array of zeros with `zeros_like`, we can use the script below:

```python
import numpy as np

a = np.zeros_like([[1, 2], [3, 4]])
```

Printing our array leaves us with this output:

```
[[0. 0.]
 [0. 0.]]
```

Chapter 3

Numba

3.1 Introduction

Numba is a Python optimization package that converts Python functions into machine code. It can be used in situations where NumPy is not optimal, such as in the integration of second order differential equations; one example would be the equation for a simple oscillating spring.

3.1.1 Example: Spring Oscillator

$$m\frac{d^2}{dt^2}x(t) = -kx(t) \tag{3.1}$$

(3.1) can be numerically evaluated for a set of initial conditions using Euler-Cromer Integration:

Algorithm 5 An Euler-Cromer integration algorithm for a simple spring oscillator

```
1: t = 0
2: while ($t \leq T$) do
3:     $a_i = -\frac{k}{m}x_{i-1}$
4:     $v_i = v_{i-1} + a_i\Delta t$
5:     $x_i = x_{i-1} + v_i\Delta t$
6:     $t = t + \Delta t$
7: end while
```

This can be written as a Python function:

```python
def integrate(k, m, x0, v0, T, dt):
    N = int(T//dt)
    t = [0]
    x = [x0]
    v = [v0]
    for n in range(N):
        a = -k*x[-1]/m
        v.append(v[-1] + a*dt)
        x.append(x[-1] + v[-1]*dt)
        t.append(t[-1]+dt)
    return t, x, v
```

If we set our parameters and initial conditions to $k = 0.1$N/m, $m = 1 \times 10^5$kg, $x_0 = 0$m, $v_0 = 2$m/s, $T = 2 \times 10^5$s, and $\Delta t = 0.01$s, it takes Python approximately 40 seconds[1] to run this loop; the result is shown in Figure 3.1.

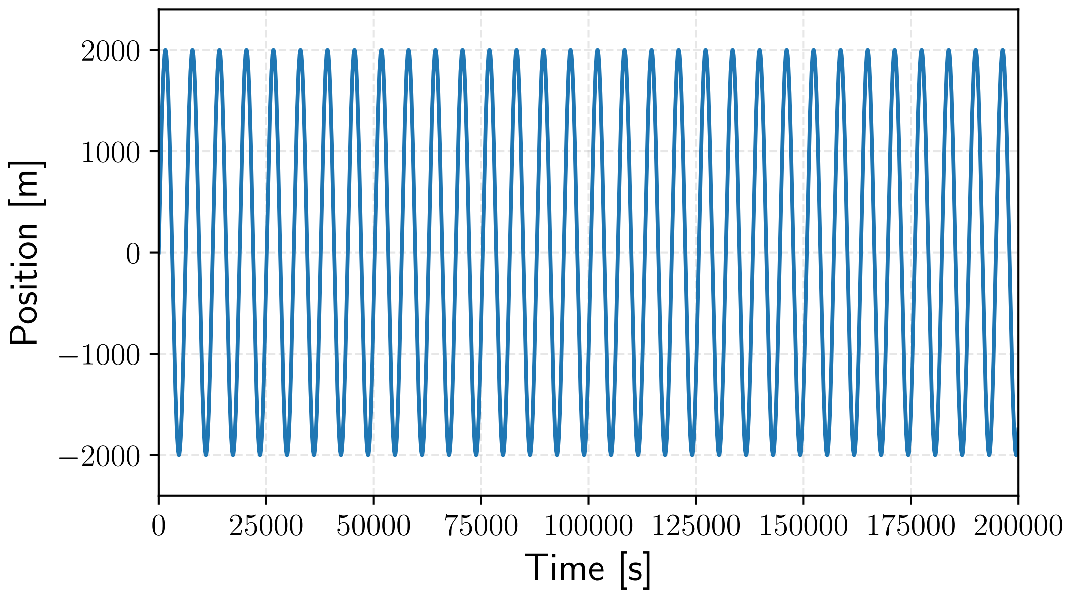


Figure 3.1: The position of a one-dimensional oscillating spring over time

Before we continue, note that we cannot vectorize this function completely since the acceleration at a given step depends on the position of the previous step; this means that NumPy is not particularly useful to us at the moment – we can however implement Numba compilation via the addition of only two lines of code:

```python
from numba import jit

@jit
def integrate(k, m, x0, v0, T, dt):
    N = int(T//dt)
    t = [0]
    x = [x0]
    v = [v0]
    for n in range(N):
        a = -k*x[-1]/m
        v.append(v[-1] + a*dt)
        x.append(x[-1] + v[-1]*dt)
        t.append(t[-1]+dt)
    return t, x, v
```

This gives us the exact same result, except this time it ran *significantly* faster – this only took 14 seconds to run when tested on the same hardware as the previous script, a 285% improvement in efficiency!

[1]The time it takes to run this program depends highly on the computer in question but bear with us in the meantime, as we are most interested in the percent increase in speed via implementing Numba.

3.2 Usage

To implement just-in-time compilation, we will need to import the `jit` function from `Numba`. To use `jit`, we simply insert a decorator just above the desired function as follows:

```python
from numba import jit

@jit                          #The decorator
def foo():                    #An arbitrary function
    return None
```

To take full advantage of the potential performance boost however, we should use the `nopython` argument; this is declared as follows:

```python
from numba import jit

@jit(nopython = True)    #The decorator in nopython mode
def foo():               #An arbitrary function
    return None
```

There are two main modes in which `@jit` can be run – `object` mode and `nopython` mode. In short, `object` mode is:

- More versatile than `nopython` mode – anything that works in standard `Python` will work in a `@jit` optimized function in `object` mode.
- Less than or equally efficient to `nopython` mode, depending on the function contents.
- Activated when setting `nopython = False`, or simply using `@jit` on its own without arguments.

While `nopython` mode is:

- More restrictive as to what the `@jit` function may contain – errors will be raised if `Numba`'s requirements aren't met!
- The most efficient way to run a program, assuming compatibility is possible.
- Activated when setting `nopython = True`.

When omitting `nopython`, `Numba` will be able to compile any given `Python` function – this is called `object` mode and is the most versatile. There is a downside to `object` mode however: in many cases, it will be slower than the alternative `nopython` mode. Generally speaking it is therefore best to use the `nopython` mode, though it comes at the cost of a lack of versatility.

The mechanism at work here is as follows: if we do not restrict our function to `nopython = True`, `Numba` will still try its best to compile as much of the function in `nopython` mode as possible, but will revert to standard `Python` code whenever it is unable to compile a part of the function.

On the other hand, using `nopython` mode prevents `Numba` from reverting to `Python` and will instead raise errors when a portion of the code cannot be compiled, ensuring that a developer can make their program as efficient as possible.

Unfortunately, learning how to debug in `nopython` mode takes time and effort, so here are a few guidelines:

- Numba supports NumPy arrays, but its support of NumPy functions is somewhat limited. [2]

- It is not possible to redefine variable types in Numba; the following function would raise an error for example:

```
@jit(nopython = True)
def foo():
    a = 20                  #Declared a variable of type int
    a = 'Hello World'       #Illegal attempt to redefine int <a> to type str
```

- Make sure the function works without the @jit decorator in the first place! You may be surprised to find that an error raised by Numba is due to an error in your Python code itself. For example, the following code snippet would raise a TypeError in standard Python, but will instead raise a Numba error due to the presence of the @jit decorator

```
@jit(nopython = True)
def foo():
    a = 1/'hello'           #Raises a Numba error, though it is a Python3 error
```

- Do not pass functions as arguments, this is not allowed. If you wish to access an external function from inside an @jit function, it must be called from within the function. It is recommended that the called function also be an @jit function.

  The following shows what *not* to do:

```
@jit(nopython = True)
def foo():
    return None

@jit(nopython = True)
def foo2(foo):
    print(foo())
```

  Instead, the following is the correct syntax:

```
@jit(nopython = True)
def foo():
    return None

@jit(nopython = True)
def foo2():
    print(foo())
```

There is a lot more to learn in this regard, and you will undoubtedly come across some error messages that appear hieroglyphic in nature. In most cases, a quick trip to StackExchange will suffice, but you may also wish to consult the official documentation for your Numba version: https://numba.pydata.org/doc.html

[2]An overview of what is supported can be found here: https://numba.pydata.org/numba-doc/dev/reference/numpysupported.html

3.3 Rates of Efficiency

Although Numba allows for the possibility of significant improvements in performance, it is important to realize that this strength manifests itself in cases where a program is running *many* iterations of an @jit function. As a result, it is possible that Numba may not be all that useful when running a program with very few iterations.

To better understand the relation between iterations and performance improvements we can create two functions with the same exact contents – the only difference between them will be that one has implemented @jit, and the other one has not:

```python
def total(N):

    '''
        Adds all integers from 0 to N and returns the resulting total sum. Also
        performs a couple more operations to make the calculations heavier.
    '''

    val = 0
    for i in range(N):
        val += i
        val /= i+1
        val *= i+1
    return val

@jit
def jit_total(N):

    '''
        Adds all integers from 0 to N and returns the resulting total sum,
        using <jit>.  Also performs a couple more operations to make the
        calculations heavier.
    '''

    val = 0
    for i in range(N):
        val += i
        val /= i+1
        val *= i+1
    return val
```

We are interested in comparing performance as a function of iterations, so by running these functions for a variety of N, timing their runtimes, and calculating their ratios, we see that the full power of Numba is not attained until a large number of iterations has occured. – this is shown in Figure 3.2:

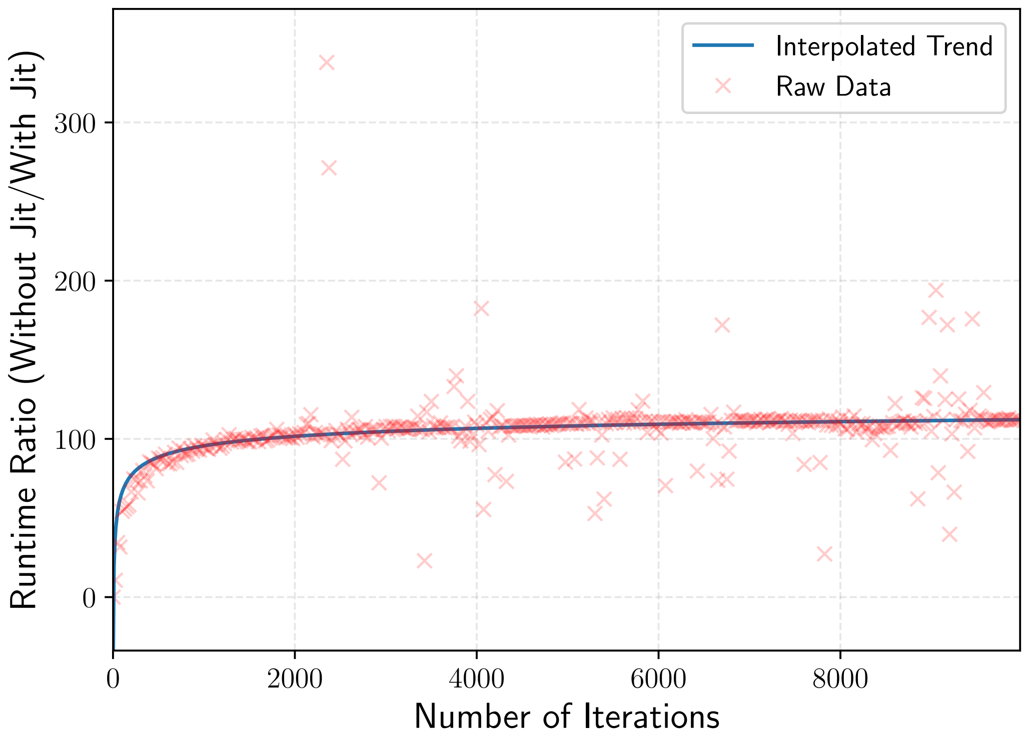


Figure 3.2: How `Numba` can affect performance based on the number of iterations taken

We can therefore conclude that using `@jit` for a small number of iterations does not give us a signification advantage in performance – however, as we increase the number of iterations, we get an increase in speed that converges to an approximate 100X speed increase[3], meaning that `@jit` is extremely useful when attempting to increase the speed of a loop, but only when the number of iterations is high enough.

[3]Results will vary depending on the function in question, of course.

Chapter 4

Example Exercises

4.1 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:

```python
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:

```python
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:

$$f(x) = x\,(\cos(x) + 1) \tag{4.1}$$

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

[1] We can see a visual interpretation of the exercise in Figure 4.2.

```python
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 4.1 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.

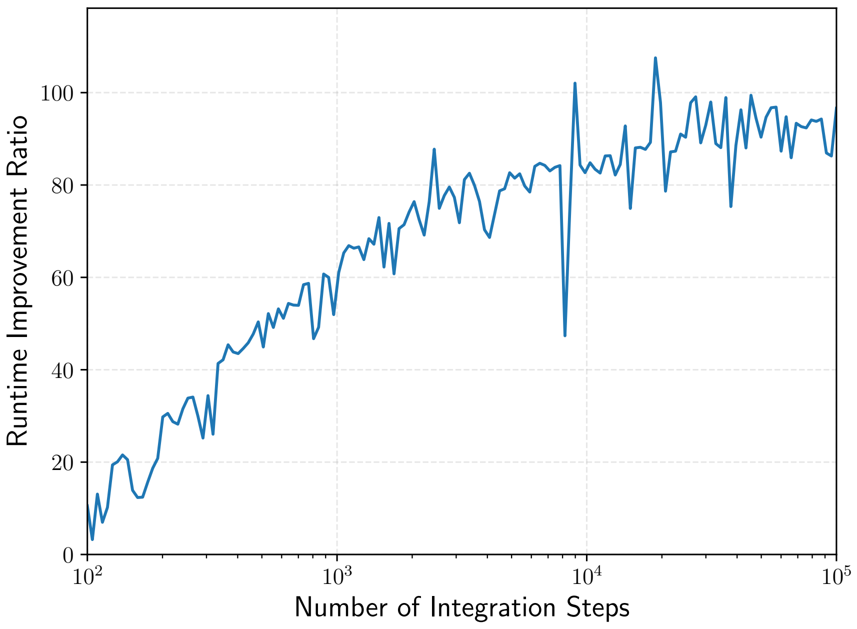


Figure 4.1: The factor by which our integration time improves by using `NumPy` for our integral.

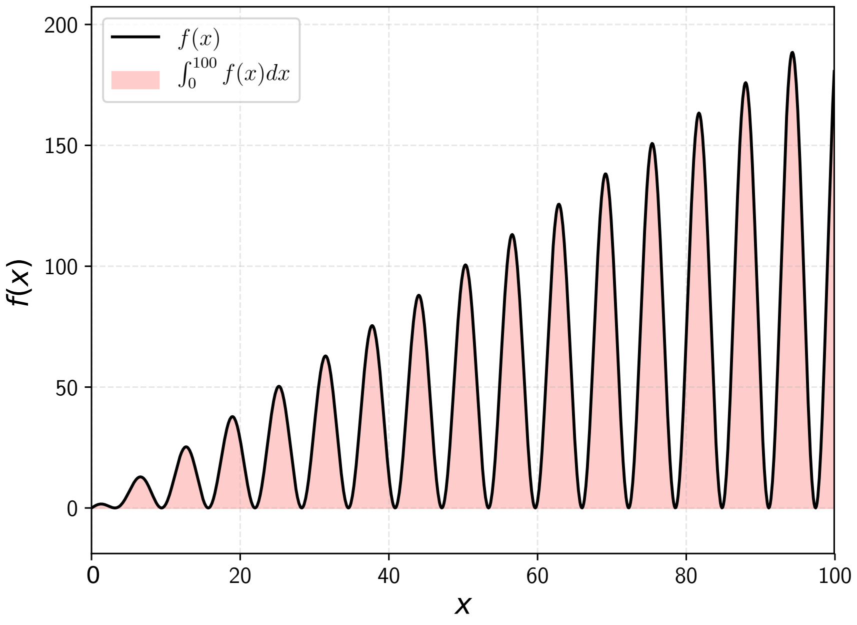


Figure 4.2: The integral $\int_0^{100} x\,(\cos(x)+1)\,dx$ visualized as the area under the curve for $f(x) = x\,(\cos(x)+1)$.

4.2 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.

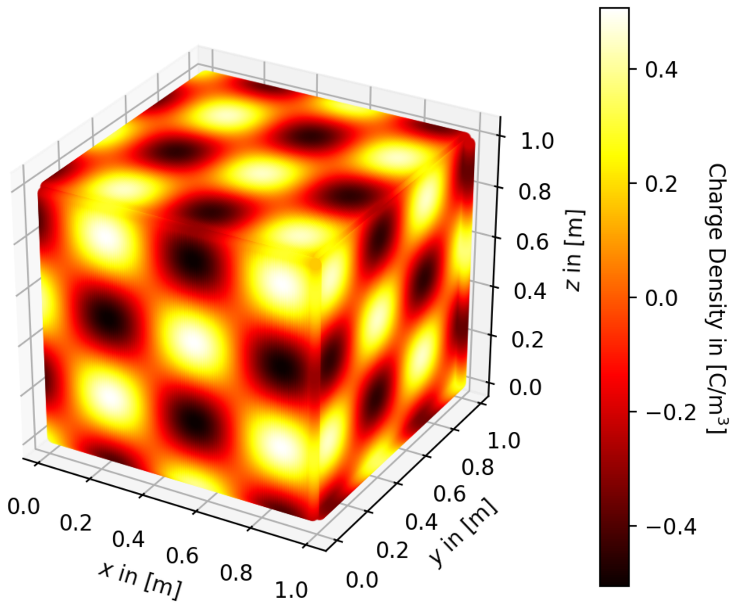


Figure 4.3: The charge density distribution ρ of a cube, visualized using a heatmap

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 4.3. The charge distribution of this cube is governed by the following equation:

$$\rho\left(x, y, z\right) = \cos\left(10(x-10)\right)\sin\left(10(y-10)\right)\cos\left(10(z-10)\right) \tag{4.2}$$

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:

$$[[x_1, y_1, z_1]\,,\ [x_2, y_2, z_2]\,,\ \ldots,\ [x_{N^3}, y_{N^3}, z_{N^3}]] \tag{4.3}$$

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:

```python
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:

```python
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:

```python
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].

4.3 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 3 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.

Bibliography

[1] L. Szacinski. (1896) Portrett av sophus lie (1842-1899). [Online]. Available: https://www.flickr.com/photos/48220291@N04/8447487830