Nanyang Technological University

Computational Physics

Y. D. Chong

The LibreTexts libraries are Powered by NICE CXOne and are supported by the Department of Education Open Textbook Pilot Project, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

TABLE OF CONTENTS

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

CHAPTER OVERVIEW

1: Scipy Tutorial

This is a tutorial for Scientific Python (Scipy), a scientific computing module for the Python programming language. There are a couple of other introductions to Scipy online, which are of excellent quality:

- Scipy Tutorial: the official tutorial.
- More Python Scientific Lecture Notes: a textbook which goes in-depth into using Scipy.

The present tutorial serves a slightly different purpose. It acts as a "walkthrough", guiding you through each step of writing a basic but complete Scipy program. You can use this as the basis for a more complete exploration of Scipy, possibly using the above online resources.

I will assume no pre-existing knowledge of the Python programming language. Programming language constructs are explained as they appear. But if you need more an even more basic tutorial on Python, feel free to consult any of the dozens available online.

1.1: Preliminaries

1.1.1 Installing Python and Scipy

If you don't have Scipy installed yet, there are plenty of installation options, detailed here.

- If you are using GNU/Linux, Python is probably already installed, so just install Scipy using your distribution's package manager (e.g. `apt-get install python3-scipy` for Debian or Ubuntu).
- If you are using Windows or Mac OS, the easiest installation method is to use the Anaconda distribution, which bundles Python with Scipy and other packages you might need. Pick the 64-bit Python 3.5 version.

From now on, I'll assume that you have installed Python 3, which is the newest version of the Python programming language. The old version, Python 2, also supports Scipy, but it brings along lots of little differences, too many and annoying to enumerate. All new (non-legacy) Python code ought to be written in Python 3.

1.1.2 Verify the Installation

If you are using GNU/Linux, open up a text terminal and type `python`. If you are using Windows, launch the program `Python 3.3 → IDLE (Python GUI)`. In each case, this will open up a text terminal with contents like this:

```
Python 3.3.3 (default, Nov 26 2013, 13:33:18)
[GCC 4.8.2] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

The `>>>` part is a command prompt. Type the following:

```
>>> from scipy import *
```

After pressing Enter, there should be a brief pause, after which you get back to the prompt. (If you see a message like `ImportError: No module named 'scipy'`, then Scipy was not installed correctly.) Next, type

```
>>> import matplotlib.pyplot as plt
```

Again, there should be no error message. These two commands initialize the Scipy scientific computing module, and the Matplotlib plotting module, so that they are now available for use in Python. Note: in the future, you don't have to type these lines in by hand when starting up Python; we'll do all the necessary "importing" commands in our program source code.

Now let's do a simple plot of $y = \sin(x)$:

```
>>> x = linspace(0, 10, 100)
>>> y = sin(x)
>>> plt.plot(x,y)
>>> plt.show()
```

This should pop up a graph showing a sine function, in a window titled "Figure 1". Here's what these four lines of code did:

1. Create an array (a sequence of numbers), consisting of 100 numbers between 0 and 10, inclusive; then give this array the name `x`.
2. Create an array whose elements are the sines of the elements in `x`; i.e., a sequence of 100 numbers, the first of which is $\sin(0)$ and the last of which is $\sin(10)$. Then, give this array the name `y`.
3. Set up an $x - y$ plot, using the `x` array as the set of x coordinates, and the `y` array as the set of y coordinates.
4. Show the plot on-screen.

If you don't understand *why* the above lines do what they do, don't worry. Let's just keep going for now.

1.2: Getting Started

1.2.1: Problem Statement: Computing Electric Potentials

Let's now walk through the steps of writing a program to perform a simple task: computing and plotting the electric potential of a set of point charges located in a one-dimensional (1D) space.

Suppose we have a set of N point charges distributed in 1D. Let us denote the positions of the particles by $\{x_0, x_1, \cdots, x_{N-1}\}$, and their electric charges by $\{q_0, q_1, \cdots, q_{N-1}$. Notice that we have chosen to start counting from zero, so that x_0 is the first position and x_{N-1} is the last position. This practice is called "zero-based indexing"; more on it later.

Knowing $\{x_j\}$ and $\{q_j\}$, we can calculate $\phi(x)$ at any arbitrary point x, by using the formula

$$\phi(x) = \sum_{j=0}^{N-1} \frac{q_j}{4\pi\epsilon_0 |x - x_j|} \tag{1.2.1}$$

The factor of $4\pi\epsilon_0$ in the denominator is annoying to keep around, so we will adopt "computational units". This means that we'll rescale the potential, positions and/or the charges so that, in the new units of measurement, $4\pi\epsilon_0 = 1$. Then the formula for the potential simplifies to

$$\phi(x) = \sum_{j=0}^{N-1} \frac{q_j}{|x - x_j|} \tag{1.2.2}$$

Our goal now is to write a computer program which takes a set of positions and charges as its input, and plots the resulting electric potential.

1.2.2: Writing into a Python Source Code File

Before writing any Python code, we should create a file to put the code in. On GNU/Linux, fire up your preferred text editor and open an empty file, naming it `potentials.py`. On Windows, in the window that was opened up by the `IDLE (Python GUI)` program, click on the menu-bar item `File` → `New File`; then type `Ctrl-s` (or click on `File` → `New File`) and save the empty file as `potentials.py`.

The file extension `.py` denotes it as a Python source code file. You should save the file in an appropriate directory.

Now let's do a very crude "first pass" at the program. Instead of handling an arbitrary number of particles, let's assume there's a *single* particle with some position x_0 and charge q_0. Then we'll plot its potential. Type the following into the file:

```
from scipy import *
import matplotlib.pyplot as plt

x0 = 1.5
q0 = 1.0

X   = linspace(-5, 5, 500)
phi = q0 / abs(X - x0)

plt.plot(X, phi)
plt.show()
```

Save the file. Now we will run the program. On GNU/Linux, open a text terminal and `cd` to the directory where you file is, then type `python -i potentials.py`. On Windows, while in file-editing window type `F5` (or click on `Run` → `Run Module`). In either case, you should see a figure pop up:

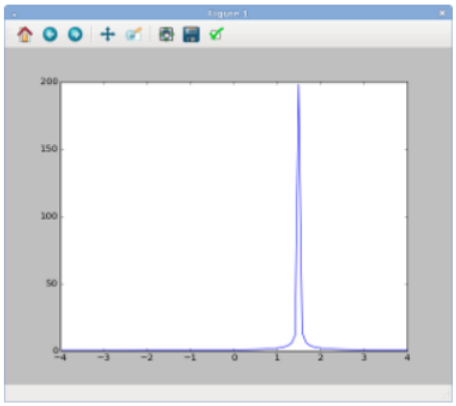

Figure 1.2.1

That's pretty much what we expect: the potential is peaked at $X = 1.5$, which is the position of the particle we specified in the program (via the variable named `x0`). The charge of the particle is given by the variable named `q0`, and we have assigned that the value 1.0. Hence, the potential is positive.

Now close the figure, and return to the Python command prompt. Note that Python is still running, even though your program has finished. From the command line, you can also examine the values of the variables which have been created by your program, simply by typing their names into the command prompt:

```
>>> x0
1.5
>>> phi
array([ 0.15384615    0.15432194    0.15480068    0.1552824  ....
          ....        0.28902404    0.28735963    0.28571429 ])
```

The value of `x0` is a number, 1.5, which was assigned to it when our program ran. The value of `phi` is more complicated: it is an *array*, which is a special data structure containing a sequence of numbers. From the command line, you can inspect the individual elements of this array. For example, to see the value of the array's first element, type this:

```
>>> phi[0]
0.153846153846
```

As we've mentioned, *index 0 refers to the first element of the array*. This so-called **zero-based indexing** is a common practice in computing. Similarly, index 1 refers to the second element of the array, index 2 refers to the third element, etc.

You can also look at the length of the array, by calling the function `len`. This function accepts an array input and returns its length, as an integer.

```
>>> len(phi)
500
```

You can exit the Python command line at any time by typing `Ctrl-d` or `exit()`.

1.3: Modularizing the Code

1.3.1: Designing a Potential Function

We could continue altering the above code in a straightforward way. For example, we could add more particles by adding variables `x1` , `x2` , `q1` , `q2` , and so forth, and altering our formula for computing `phi` . However, this is not very satisfactory: each time we want to consider a new collection of particle positions or charges, or change the number of particles, we would have to re-write the program's internal "logic"—i.e., the part that computes the potentials. In programming terminology, our program is insufficiently "modular". Ideally, we want to isolate the part of the program that computes the potential from the part that specifies the numerical inputs to the calculation, like the positions and charges.

To modularize the code, let's define a function that computes the potential of an *arbitrary* set of charged particles, sampled at an *arbitrary* set of positions. Such a function would need three sets of inputs:

- An array of particle positions $\vec{x} \equiv [x_0, \cdots, x_{N-1}]$. (Don't get confused, by the way: we are using these N numbers to refer to the positions of N particles in a 1D space, *not* the position of a single particle in an N-dimensional space.)
- An array of particle charges $\vec{q} \equiv [q_0, \cdots, q_{N-1}]$.
- An array of sampling points $\vec{X} \equiv [X_0, \cdots, X_{M-1}]$, which are the points where we want to know $\phi(X)$.

The number of particles, N, and the number of sampling points, M, should be arbitrary positive integers. Furthermore, N and M need not be equal.

The function we intend to write must compute the array

$$\begin{bmatrix} \phi(X_0) \\ \phi(X_1) \\ \vdots \\ \phi(X_{M-1}) \end{bmatrix} \tag{1.3.1}$$

which contains the value of the total electric potential at each of the sampling points. The total potential can be written as the sum of contributions from all particles. Let us define $\phi_j(x)$ as the potential produced by particle j:

$$\phi_{j(x)} \equiv \frac{q_j}{|x - x_j|} \tag{1.3.2}$$

Then the total potential is

$$\begin{bmatrix} \phi(X_0) \\ \phi(X_1) \\ \vdots \\ \phi(X_{M-1}) \end{bmatrix} = \begin{bmatrix} \phi_0(X_0) \\ \phi_0(X_1) \\ \vdots \\ \phi_0(X_{M-1}) \end{bmatrix} + \begin{bmatrix} \phi_1(X_0) \\ \phi_1(X_1) \\ \vdots \\ \phi_1(X_{M-1}) \end{bmatrix} + \cdots + \begin{bmatrix} \phi_{N-1}(X_0) \\ \phi_{N-1}(X_1) \\ \vdots \\ \phi_{N-1}(X_{M-1}) \end{bmatrix}. \tag{1.3.3}$$

1.3.2 Writing the Program

Let's code this up. Return to the file `potentials.py` , and **delete the entire contents of the file**. Then replace it with the following:

```
from scipy import *
import matplotlib.pyplot as plt

## Return the potential at measurement points X, due to particles
## at positions xc and charges qc.  xc, qc, and X must be 1D arrays,
## with xc and qc of equal length.  The return value is an array
## of the same length as X, containing the potentials at each X point.
def potential(xc, qc, X):
    M = len(X)
```

```
    N = len(xc)
    phi = zeros(M)
    for j in range(N):
        phi += qc[j] / abs(X - xc[j])
    return phi

charges_x  = array([0.2, -0.2])
charges_q  = array([1.5, -0.1])
xplot = linspace(-3, 3, 500)

phi = potential(charges_x, charges_q, xplot)

plt.plot(xplot, phi)
pmin, pmax = -50, 50
plt.ylim(pmin, pmax)
plt.show()
```

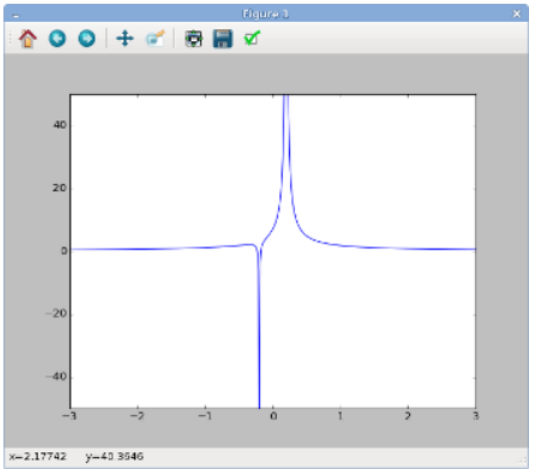

Figure 1.3.1

When typing or pasting the above into your file, be sure to preserve the **indentation** (i.e., the number of spaces at the beginning of each line). Indentation is important in Python; as we'll see, it's used to determine program structure. Now save and run the program again:

- In the Windows GUI, type `F5` in the editing window showing `potentials.py` .
- On GNU/Linux, type `python -i potentials.py` from the command line.
- Alternatively, from the Python command line, type `import potentials` , which will load and run your `potentials.py` file.

You should now see a figure like the one on the right, showing the electric potential produced by two particles, one at position $x_0 = 0.2$ with charge $q_0 = 1.5$ and the other at position $x_1 = -0.2$ with charge $q_1 = -0.1$.

There are less than 20 lines of actual code in the above program, but they do quite a lot of things. Let's go through them in turn:

Module Imports

The first two lines import the Scipy and Matplotlib modules, for use in our program. We have not yet explained how importing works, so let's do that now.

```
from scipy import *
import matplotlib.pyplot as plt
```

Each Python module, including Scipy and Matplotlib, defines a variety of functions and variables. If you use multiple modules, you might have a situation where, say, two different modules each define a function with the same name, but doing entirely different things. That would be Very Bad. To help avoid this, Python implements a concept called a **namespace**. Suppose you import a module (say Scipy) like this:

```
import scipy
```

One of the functions defined by Scipy is `linspace`, which we have already seen. This function was defined by the `scipy` module, and lies inside the `scipy` namespace. As a result, when you import the Scipy module using the `import scipy` line, you have to call the `linspace` function like this:

```
x = scipy.linspace(-3, 3, 500)
```

The `scipy.` in front says that you're referring to the `linspace` function that was defined in the `scipy` namespace. (Note: the online documentation for `linspace` refers to it as `numpy.linspace`, but the exact same function is also present in the `scipy` namespace. In fact, all `numpy.*` functions are replicated in the `scipy` namespace. So unless stated otherwise, we only have to import `scipy`.)

We will be using a lot of functions that are defined in the `scipy` namespace. Since it would be annoying to have to keep typing `scipy.` all over the place, we opt to use a slightly different import statement:

```
from scipy import *
```

This imports all the functions and variables in the `scipy` namespace directly into your program's namespace. Therefore, you can just call `linspace`, without the `scipy.` prefix. Obviously, you don't want to do this for every module you use, otherwise you'll end up with the name-clashing problem we alluded to earlier! The only module we'll use this shortcut with is `scipy`.

Another way to avoid having to type long prefixes is shown by this line:

```
import matplotlib.pyplot as plt
```

This imports the `matplotlib.pyplot` module (i.e., the `pyplot` module which is nested inside the `matplotlib` module). That's where `plot`, `show`, and other plotting functions are defined. The `as plt` in the above line says that we will refer to the `matplotlib.pyplot` namespace as the short form `plt` instead. Hence, instead of calling the `plot` function like this:

```
matplotlib.pyplot.plot(x, y)
```

we will call it like this:

```
plt.plot(x, y)
```

Comments

Let's return to the program we were looking at earlier. The next few lines, beginning with `#`, are "comments". Python ignores the `#` character and everything that follows it, up to the end of the line. Comments are very important, even in simple programs like this.

When you write your own programs, please remember to include comments. You don't need a comment for every line of code—that would be excessive—but at a minimum, each function should have a comment explaining what it does, and what the inputs and return values are.

Function Definition

Now we get to the *function definition* for the function named `potential`, which is the function that computes the potential:

```
def potential(xc, qc, X):
    M = len(X)
    N = len(xc)
    phi = zeros(M)
    for j in range(N):
        phi += qc[j] / abs(X - xc[j])
    return phi
```

The first line, beginning with `def`, is a **function header**. This function header states that the function is named `potential`, and that it has three inputs. In computing terminology, the inputs that a function accepts are called **parameters**. Here, the parameters are named `xc`, `qc` and `X`. As explained by the comments, we intend to use these for the positions of the particles, the charges of the particles, and the positions at which to measure the potential, respectively.

The function definition consists of the function header, together the rest of the *indented* lines below it. The function definition terminates once we get to a line which is at the same indentation level as the function header. (That terminating line is considered a separate line of code, which is not part of the function definition).

By convention, you should use 4 spaces per indentation level.

The indented lines below the function header are called the **function body**. This is the code that is run each time the function is called. In this case, the function body consists of six lines of code, which are intended to compute the total electric potential, according to the procedure that we have outlined in the preceding section:

- The first two lines define two helpful variables, `M` and `N`. Their values are set to the lengths of the `X` and `xc` arrays, respectively.
- The next line calls the `zeros` function. The input to `zeros` is `M`, the length of the `X` array (i.e., our function's third parameter). Therefore, `zeros` returns an array, of the same same length as `X`, with every element set to 0.0. For now, this represents the electric potential in the absence of any charges. We give this array the name `phi`.
- The function then iterates over each of the particles and add up its contribution to the potential, using a construct known as a `for` loop. The code `for j in range(N):` is the loop's "header line", and the next line, indented 4 spaces deeper than the header line, is the "body" of the loop.

  The header line states that we should run the loop body several times, with the variable `j` set to different values during each run. The values of `j` to loop over are given by `range(N)`. This is a function call to the range function, with `N` (the number of electric charges) as the input. The `range(N)` function call returns a sequence specifying `N` successive integers, from 0 to `N-1`, inclusive. (Note that the last value in the sequence is `N-1`, not `N`. Because we start from 0, this means that there is a total of `N` integers in the sequence. Also, calling `range(N)` is the same as calling `range(0,N)`.)
- For each `j`, we compute `qc[j] / abs(X - xc[j])`. This is an array whose elements are the values of the electric potential at the set of positions `X`, arising from the individual particle$j$. In mathematical terms, we are calculating

$$\phi_j(X) \equiv \frac{q_j}{|X - x_j|} \tag{1.3.4}$$

  using the array of positions `X`. We then add this array to `phi`. Once this is done for all `j`, the array `phi` will contain the desired total potential,

$$\phi(X) = \sum_{j=0}^{N-1} \phi_j(X). \tag{1.3.5}$$

- Finally, we call `return` to specify the function's output, or **return value**. This is the array `phi`.

Top-Level Code: Numerical Constants

After the function definition comes the code to use the function:

```
charges_x  = array([0.2, -0.2])
charges_q  = array([1.5, -0.1])
xplot = linspace(-3, 3, 500)
```

Like the import statements at the beginning of the program, these lines of code lie at **top level**, i.e., they are not indented. The function header which defines the `potential` function is also at top level. Running a Python program consists of running its top level code, in sequence.

The above lines define variables to store some numerical constants. In the first two lines, `charges_x` and `charges_q` variables store the numerical values of the positions and charges we are interested in. These are initialized using the `array` function. You may be wondering why the `array` function call has square brackets nested in commas. We'll explain later, in part 2 of the tutorial.

On the third line, the `linspace` function call returns an array, whose contents are initialized to the 500 numbers between -3 and 3 (inclusive).

Next, we call the `potential` function, passing `charges_x` , `charges_q` and `xplot` as the inputs:

```
phi = potential(charges_x, charges_q, xplot)
```

These inputs provide the values of the function definition's parameters `xc` , `qc` , and `X` respectively. The return value of the function call is an array containing the total potential, evaluated at each of the positions specified in `xplot` . This return value is stored as the variable named `phi` .

Plotting

Finally, we create the plot:

```
plt.plot(xplot, phi)
pmin, pmax = -50, 50
plt.ylim(pmin, pmax)
plt.show()
```

We have already seen how the `plot` and `show` functions work. Here, prior to calling `plt.show` , we have added two extra lines to make the potential curve is more legible, by adjust the plot's y-axis bounds. The ylim function accepts two parameters, the lower and upper bounds of the y-axis. In this case, we set the bounds to -50 and 50 respectively. There is an xlim function to do the same for the x-axis.

Notice that in the line `pmin, pmax = -50, 50` , we set two variables (`pmin` and `pmax`) on the same line. This is a little "syntactic sugar" to make the code a little easier to read. It's equivalent to having two separate lines, like this:

```
pmin = -50
pmax = 50
```

We'll explain how this construct works in the next part of the tutorial.

CHAPTER OVERVIEW

2: Scipy Tutorial (Part 2)

This is part 2 of the Scientific Python tutorial.

2.1: Sequential Data Structures

In the previous part of the tutorial, we worked through a simple example of a Scipy program which calculates the electric potential produced by a collection of charges in 1D. At the time, we did not explain much about the data structures that we were using to store numerical information (such as the values of the electric potential at various points). Let's do that now.

There are three common data structures that will be used in a Scipy program for scientific computing: arrays, lists, and tuples. These structures store linear sequences of Python objects, similar to the concept of "vectors" in physics and mathematics. However, these three types of data structures all have slightly different properties, which you should be aware of.

2.1.1 Arrays

An array is a data structure that contains a sequence of numbers. Let's do a quick recap. From a fresh Python command prompt, type the following:

```
>>> from scipy import *
>>> x = linspace(-0.5, 0.5, 9)
>>> x
array([-0.5  , -0.375, -0.25 , -0.125,  0.   ,  0.125,  0.25 ,  0.375,  0.5  ])
```

The first line, as usual, is used to import the `scipy` module. The second line creates an array named `x` by calling linspace, which is a function defined by `scipy`. With the given inputs, the function returns an array of 9 numbers between -0.5 and 0.5, inclusive. The third line shows the resulting value of `x`.

The array data structure is provided specifically by the Scipy scientific computing module. Arrays can only contain numbers (furthermore, each individual array can only contain numbers of one type, e.g. integers or complex numbers; we'll discuss this in the next article). Arrays also support special facilities for doing numerical linear algebra. They are commonly created using one of these functions from the `scipy` module:

- linspace, which creates an array of evenly-spaced values between two endpoints.
- arange, which creates an array of integers in a specified range.
- zeros, which creates an array whose elements are all 0.0.
- ones, which creates an array whose elements are all 1.0.
- empty, which creates an array whose elements are uninitialized (this is usually used when you want to set the elements later).

Of these, we've previously seen examples of the `linspace` and `zeros` functions being used. As another example, to create an array of 500 elements all containing the number -1.2, you can use the `ones` function and a multiplication operation:

```
x = -1.2 * ones(500)
```

An alternative method, which is *slightly* faster, is to generate the array using `empty` and then use the `fill` method to populate it:

```
x = empty(500); x.fill(-1.2)
```

One of the most important things to know about an array is that its size is fixed at the moment of its creation. When creating an array, you need to specify exactly how many numbers you want to store. If you ever need to revise this size, you must create a *new* array, and transfer the contents over from the old array. (For very big arrays, this might be a slow operation, because it involves copying a lot of numbers between different parts of the computer memory.)

You can pass arrays as inputs to functions in the usual way (e.g., by supplying its name as the argument to a function call). We have already encountered the `len` function, which takes an array input and returns the array's length (an integer). We have also encountered the `abs` function, which accepts an array input and returns a new array containing the corresponding absolute values. Similar to `abs`, many mathematical functions and operations accept arrays as inputs; usually, this has the effect of applying the function or operation to each element of the input array, and returning the result as another array. The returned array

has the same size as the input array. For example, the `sin` function with an array input `x` returns another array whose elements are the sines of the elements of `x` :

```
>>> y = sin(x)
>>> y
array([-0.47942554, -0.36627253, -0.24740396, -0.12467473,  0.        ,
        0.12467473,  0.24740396,  0.36627253,  0.47942554])
```

You can access individual elements of an array with the notation `a[j]` , where `a` is the variable name and `j` is an integer index (where the first element has index 0, the second element has index 1, etc.). For example, the following code sets the first element of the `y` array to the value of its second element:

```
>>> y[0] = y[1]
>>> y
array([-0.36627253, -0.36627253, -0.24740396, -0.12467473,  0.        ,
        0.12467473,  0.24740396,  0.36627253,  0.47942554])
```

Negative indices count backward from the end of the array. For example:

```
>>> y[-1]
0.47942553860420301
```

Instead of setting or retrieving individual values of an array, you can also set or retrieve a *sequence* of values. This is referred to as **slicing**, and is described in detail in the Scipy documentation. The basic idea can be demonstrated with a few examples:

```
>>> x[0:3] = 2.0
>>> x
array([ 2.   ,  2.   ,  2.   , -0.125,  0.   ,  0.125,  0.25 ,  0.375,  0.5  ])
```

The above code accesses the elements in array `x` , starting from index 0 up to but not including 3 (i.e. indices 0, 1, and 2), and assigns them the value of `2.0` . This changes the contents of the array `x` .

```
>>> z = x[0:5:2]
>>> z
array([ 2.,  2.,  0.])
```

The above code retrieves a subset of the elements in array `x` , starting from index 0 up to but not including 5, and stepping by 2 (i.e., the indices 0, 2, and 4), and then groups those elements into an array named `z` . Thereafter, `z` can be treated as an array.

Finally, arrays can also be multidimensional. If we think of an ordinary (1D) array as a vector, then a 2D array is equivalent to a matrix, and higher-dimensional arrays are like tensors. We will see practical examples of higher-dimensional arrays later. For now, here is a simple example:

```
>>> y = zeros((4,2))     # Create a 2D array of size 4x2
>>> y[2,0] = 1.0
>>> y[0,1] = 2.0
>>> y
array([[ 0.,  2.],
       [ 0.,  0.],
```

```
       [ 1.,  0.],
       [ 0.,  0.]])
```

2.1.2: Lists

There is another type of data structure called a **list**. Unlike arrays, lists are built into the Python programming language itself, and are not specific to the Scipy module. Lists are general-purpose data structures which are *not* optimized for scientific computing (for example, we will need to use arrays, not lists, when we want to do linear algebra).

The most convenient thing about Python lists is that you can specify them explicitly, using `[...]` notation. For example, the following code creates a list named `u`, containing the integers 1, 1, 2, 3,and 5:

```
>>> u = [1, 1, 2, 3, 5]
>>> u
[1, 1, 2, 3, 5]
```

This way of creating lists is also useful for creating Scipy arrays. The array function accepts a list as an input, and returns an array containing the same elements as the input list. For example, to create an array containing the numbers 0.2, 0.1,and 0.0:

```
>>> x = array([0.2, 0.1, 0.0])
>>> x
array([ 0.2,  0.1,  0. ])
```

In the first line, the square brackets create a list object containing the numbers 0.2, 0.1,and 0.0, then passes that list directly as the input to the `array` function. The above code is therefore equivalent to the following:

```
>>> inputlist = [0.2, 0.1, 0.0]
>>> inputlist
[0.2, 0.1, 0.0]
>>> x = array(inputlist)
>>> x
array([ 0.2,  0.1,  0. ])
```

Usually, we will do number crunching using arrays rather than lists. However, sometimes it is useful to work directly with lists. One convenient thing about lists is that they can contain arbitrary Python objects, of any data type; by contrast, arrays are allowed only to contain numerical data.

For example, a Python list can store character strings:

```
>>> u = [1, 2, 'abracadabra', 3]
>>> u
[1, 2, 'abracadabra', 3]
```

And you can set or retrieve individual elements of a Python list in the same way as an array:

```
>>> u[1] = 0
>>> u
[1, 0, 'abracadabra', 3]
```

Another great advantage of lists is that, unlike arrays, you can dynamically increase or decrease the size of a list:

```
>>> u.append(99)                    # Add 99 to the end of the list u
>>> u.insert(0, -99)                # Insert -99 at the front (index 0) of the list u
>>> u
[-99, 1, 0, 'abracadabra', 3, 99]
>>> z = u.pop(3)                    # Remove element 3 from list u, and name it z
>>> u
[-99, 1, 0, 3, 99]
>>> z
'abracadabra'
```

Aside: About Methods

In the above example, `append`, `insert`, and `pop` are called **methods**. You can think of a method as a special kind of function which is "attached" to an object and relies upon it in some way. Just like a function, a method can accept inputs, and give a return value. For example, when `u` is a list, the code

```
z = u.pop(3)
```

means to take the list `u`, find element index 3 (specified by the input to the method), remove it from the list, and return the removed list element. In this case, the returned element is named `z`. See here for a summary of Python list methods. We'll see more examples of methods as we go along.

2.1.3 Tuples

Apart from lists, Python provides another kind of data structure called a **tuple**. Whereas lists can be constructed using square bracket `[...]` notation, tuples can be constructed using parenthetical `(...)` notation:

```
>>> v = (1, 2, 'abracadabra', 3)
>>> v
(1, 2, 'abracadabra', 3)
```

Like lists, tuples can contain any kind of data type. But whereas the size of a list can be changed (using methods like `append`, `insert`, and `pop`, as described in the previous subsection), the size of a tuple is fixed once it's created, just like an array.

Tuples are mainly used as a convenient way to "group" or "ungroup" named variables. Suppose we want to split the contents of `v` into four separate named variables. We could do it like this:

```
>>> dog, cat, apple, banana = v
>>> dog
1
>>> cat
2
>>> apple
'abracadabra'
>>> banana
3
```

On the left-hand side of the `=` sign, we're actually specifying a tuple of four variables, named `dog`, `cat`, `apple`, and `banana`. In cases like this, it is OK to omit the parentheses; when Python sees a group of variable names separated by commas, it automatically treats that group as a tuple. Thus, the above line is equivalent to

```
>>> (dog, cat, apple, banana) = v
```

We saw a similar example in the previous part of the tutorial, where there was a line of code like this:

```
pmin, pmax = -50, 50
```

This assigns the value -50 to the variable named `pmin`, and 50 to the variable named `pmax`. We'll see more examples of tuple usage as we go along.

2.2: Improving the Program

Let's return to the program for calculating the electric potential, which we discussed in the previous part of the tutorial, and improve it further. These improvements will show off some more advanced features of Python and Scipy which are good to know about.

We'll also make one substantive change in the physics: instead of treating the particles as point-like objects, we'll assume that they have a finite radius R, with all the charge concentrated at the surface. Hence, the potential produced by a particle of total charge q_0 and position x_0 will have the form

$$\phi(X) = \begin{cases} \dfrac{q_0}{|X - x_0|}, & \text{if } |X - x_0| \geq R \\ \dfrac{q_0}{R} & \text{if } |X - x_0| < R. \end{cases} \tag{2.2.1}$$

Open a few Python file, and call it `potentials2.py` . Write the following into it:

```
from scipy import *
import matplotlib.pyplot as plt

## Return the potential at measurement points X, due to particles
## at positions xc and charges qc.  xc, qc, and X must be 1D arrays,
## with xc and qc of equal length.  The return value is an array
## of the same length as X, containing the potentials at each X point.
def potential(xc, qc, X, radius=5e-2):
    assert xc.ndim == qc.ndim == X.ndim == 1
    assert len(xc) == len(qc)
    assert radius > 0.

    phi = zeros(len(X))
    for j in range(len(xc)):
        dphi = qc[j] / abs(X - xc[j])
        dphi[abs(X - xc[j]) < radius] = qc[j] / radius
        phi += dphi
    return phi

## Plot the potential produced by N particles of charge 1, distributed
## randomly between x=-1 and x=1.
def potential_demo(N=20):
    X = linspace(-2.0, 2.0, 200)
    qc = ones(N)

    from scipy.stats import uniform
    xc = uniform(loc=-1.0, scale=2.0).rvs(size=N)

    phi = potential(xc, qc, X)

    fig_label = 'Potential from ' + str(N) + ' particles'
    plt.plot(X, phi, 'ro', label=fig_label)
    plt.ylim(0, 1.25 * max(phi))
```

```
    plt.legend()
    plt.xlabel('r')
    plt.ylabel('phi')
    plt.show()

potential_demo(100)
```

We will now go through the changes that we've made in the program.

2.2.1 Optional Function Parameters

As before, we define a function named `potential`, whose job is to compute the electric potential produced by a collection of particles in 1D. However, you might notice a change in the function header:

```
def potential(xc, qc, X, radius=5e-2):
```

We have added an **optional parameter**, specified as `radius=5e-2`. An optional parameter is a parameter which has a default value. In this case, the optional parameter is named `radius`, and its default value is `5e-2` (which means 5×10^{-2}; you can also write it as `0.05`, which is equivalent). If you call the function omitting the last input, the value will be assumed to be 0.05. If you supply an explicit value for the last input, that overrides the default.

If a function call omits a *non*-optional parameter (which as `xc`), that is a fatal error: Python will stop the program with an error message.

2.2.2 Assert Statements

In the function body, we have added the following three lines:

```
    assert xc.ndim == qc.ndim == X.ndim == 1
    assert len(xc) == len(qc)
    assert radius > 0.
```

The `assert` statement is a special Python statement which checks for the truth value of the following expression; if that expression is false, the program will stop and an informative error message will be displayed.

Here, we use the `assert` statements to check that

- `xc`, `qc`, and `X` are all 1D arrays (note: the `==` Python operator checks for numerical equality)
- `xc` has the same length as `qc`
- `radius` has a positive value (note: `0.` is Python short-hand for the number `0.0`).

Similar to writing comments, adding `assert` statements to your program is good programming practice. They are used to verify that the assumptions made by the rest of the code (e.g., that the `xc` and `qc` arrays have equal length) are indeed met. This ensures that if we make a programming mistake (e.g., supplying arrays of incompatible size as inputs), the problem will surface as soon as possible, rather than letting the program continue to run and causing a more subtle error later on.

2.2.3 Advanced Slicing

Inside the for loop, we have changed the way the potential is computed:

```
    for j in range(len(xc)):
        dphi = qc[j] / abs(X - xc[j])
        dphi[abs(X - xc[j]) < radius] = qc[j] / radius
        phi += dphi
```

As discussed above, we are now considering particles of finite size rather than point particles, so the potential is constant at distances below the particle radius. This is accomplished using an advanced array slicing technique.

For each particle j, the potential is computed in three steps:

- Calculate the potential using the regular formula $q_j/|X-x_j|$, and save those values into an array, one for each value of X.
- Find the indices of that array which correspond to values with $|X-x_j|<R$, and overwrite those elements with the constant value q_j/R. To find the relevant indices, we make use of the following slicing feature: if a comparison expression is supplied as an index, that refers to those indices for which the comparison is true. In this case, the comparison expression is `abs(X-xc[j]) < radius`, which refers to the indices of X which are below the minimum radius. These indices are the ones in the `dphi` array that we want to overwrite.
- Add the result to the total potential.

Demo Function

Finally, we have a "demo" or ("demonstration") function to make the appropriate plots:

```
## Plot the potential produced by N particles of charge 1, distributed
## randomly between x=-1 and x=1.
def potential_demo(N=20):
    X = linspace(-2.0, 2.0, 200)
    qc = ones(N)

    from scipy.stats import uniform
    xc = uniform(loc=-1.0, scale=2.0).rvs(size=N)

    phi = potential(xc, qc, X)

    fig_label = 'Potential from ' + str(N) + ' particles'
    plt.plot(X, phi, 'ro', label=fig_label)
    plt.ylim(0, 1.25 * max(phi))
    plt.legend()
    plt.xlabel('r')
    plt.ylabel('phi')
    plt.show()

potential_demo(100)
```

Whereas our previous program put the plotting stuff at "top level", here we encapsulate the plotting code in a `potential_demo()` function. This function is called by the top-level statement `potential_demo(100)`, which occurs at the very end of the program.

It is useful to do this because if, in the future, you want the program demonstrate something else (e.g. producing a different kind of plot), it won't be necessary to delete the `potential_demo` function (and risk having to rewrite it if you change your mind). Instead, you can write another demo function, and revise that single top-level statement to call the new demo function instead.

The `potential_demo` function provides another example of using optional parameters. It accepts a parameter `N=20`, specifying the number of particles to place. When the program runs, however, the function is invoked through the top-level statement `potential_demo(100)`, i.e. with an actual input of 100 which overrides the default value of 20. If the top-level statement had instead been `potential_demo()`, then the default value of 20 would be used.

Sampling Random Variables

The demo function generates `N` particles with random positions. This is done using this code:

```
    from scipy.stats import uniform
    xc = uniform(loc=-1.0, scale=2.0).rvs(size=N)
```

The first line imports a function named `uniform` from the `scipy.stats` module, which is a module that implements random number distributions. As this example shows, import statements don't have to be top-level statements. In some cases, we might choose to perform an import only when a particular function runs (usually, this is done if that function is the only one in the program relying on that module).

The `uniform` function returns an object which corresponds to a particular uniform distribution. One of the methods of this object, named `rvs`, generates an array of random numbers drawn from that distribution.

Plotting

After computing the total potential using a call to the `potential` function, we plot it:

```
    fig_label = 'Potential from ' + str(N) + ' particles'
    plt.plot(X, phi, 'ro', label=fig_label)
```

To begin with, concentrate on the second line. This is a slightly more sophisticated use of Matplotlib's `plot` function than what we had the last time.

The first two arguments, as before, are the x and y coordinates for the plot. The next argument, `'ro'`, specifies that we want to plot using red circles, rather than using lines with the default color.

The fourth argument, `label=fig_label`, specifies some text with which to label the plotted curve. It is often useful to associate each curve in a figure with a label (though, in this case, the figure contains only one curve).

This way of specifying a function input, which has the form `FOO=BAR`, is something we have not previously seen. It relies on a feature known as **keyword arguments**. In this case, `label` is the keyword (the name of the parameter we're specifying), and `fig_label` is the value (which is a string object; we'll discuss this below). Keyword arguments allow the caller of a function to specify optional parameters in any order. For example,

```
    plt.plot(X, phi, 'ro', label=fig_label, linewidth=2)
```

is equivalent to

```
    plt.plot(X, phi, 'ro', linewidth=2, label=fig_label)
```

The full list of keywords for the `plot` function is given is its documentation.

Constructing a Label String

Next, we turn to the line

```
    fig_label = 'Potential from ' + str(N) + ' particles'
```

which creates a Python object named `fig_label`, which is used for labeling the curve. This kind of object is called a **character string** (or just **string** for short).

On the right-hand side of the above statement, we build the contents of the string from several pieces. This is done in order to get a different string for each value of `N`. The `+` operator "concatenates" strings, joining the strings to its left and right into a longer string. In this case, the string `fig_label` consists of the following shofter strings, concatenated together:

- A string containing the text `'Potential from '`.
- A string containing the numerical value of `N`, in text form. This is computed using the `str` function, which converts numbers into their corresponding string representations.
- A string containing the text `' particles'`.

The rest of the `potential_demo` function is relatively self-explanatory. The `ylim` function specifies the lower and upper limits of the plot's y-axis (there is a similar `xlim` function, which we didn't use). The `plt.legend()` statement causes the curve label to be shown in a legend included in the plot. Finally, the `xlabel` and `ylabel` functions add string labels to the x and y axes.

Running the Program

Now save your `potential2.py` program, and run it. (Reminder: you can do this by typing `python -i potential2.py` from the command line on GNU/Linux, or `F5` in the Windows GUI, or `import potential2` from the Python command line). The resulting figure looks like this:

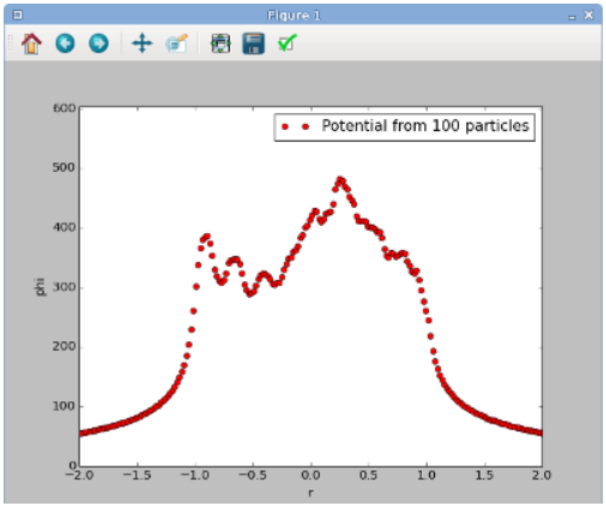

Figure 2.2.1

CHAPTER OVERVIEW

3: Numbers, Arrays, and Scaling

In this article, we will cover the basic concepts of computation, and explain how they relate to writing programs for scientific computing.

3.1: A Model of Computing

A modern computer is a tremendously complex system, with many things to understand at the level of the hardware, the operating system, and the programming software (e.g. Python). It is helpful to consider a simplified **computing model**, which is basically a "cartoon" representation of a computer that omits the unimportant facts about how it operates, and focuses on the most important aspects of what a computer program does.

The standard paradigm of computing that we use today is the Von Neumann architecture, which divides a computer into three inter-connected units: processor, memory, and input/output devices. We'll use this as the basis of our simplified model, with an emphasis on the processor and memory parts.

A computer's **memory** is essentially a chunk of space where we can store numbers. For now, we won't concern ourselves with how the contents of memory are organized or formatted. The **processor** can read one or more numbers from any locations (or **addresses**) in memory, perform some basic operations on them, and then write the results into any other addresses. We also make three other important assumptions:

1. The capacity is effectively infinite; we don't worry about running out.
2. The memory is **random-access memory** (RAM), meaning that the processor can access *any* addresses in memory, one after another, with the same speed. (In real life, not all types of memory are random-access. Disk drives, for instance, are not, because the scanning head must physically move to different positions in order to read different parts of memory. However, ordinary computer programs can ignore such details, which are left to the operating system to manage.)
3. The processor can only do one thing at a time. For example, if you ask it to read two numbers from memory, that takes twice as long as reading a single number from memory. (Again, real computers violate this assumption to some extent; computers now commonly have multiple processors that can performs multiple operations simultaneously. Our present simple model ignores these complications.)

A **program** is a set of instructions for the processor. For example, the following line of code is a program (or part of a program) that tells the processor to add up four numbers that are currently stored in memory, and save the result to another memory address labeled `x`:

```
x = a + b + c + d
```

Because the processor can only do one thing at a time, even a simple line of code like this involves several sequential steps. The processor can't simultaneously read all four memory addresses (corresponding to `a` through `d`); it must read them one at a time. The following figure shows how the processor might carry out the above addition program:

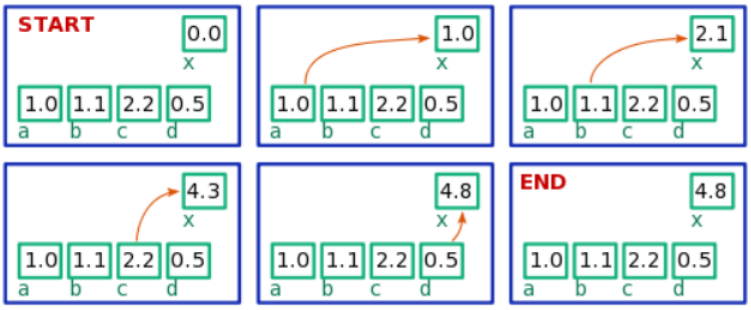

Figure 3.1.1: A sequence of steps for adding up four numbers, `x = a + b + c + d`. The computer's memory is visualized as a blue box; the variables `x` and `a` through `d` correspond to addresses in memory, shown as smaller green boxes containing numbers. The processor performs the additions one at a time.

3.2: Integers and Floating-Point Numbers

Digital computers store all data in the form of bits (ones and zeros), and a number is typically stored as a sequence of bits *of fixed length*. For example, a number labeled `x` might refer to a sequence of eight bits starting from some specific address, as shown in the following figure:

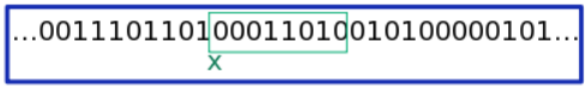

Figure 3.2.1: An eight-bit number stored in a variable `x`.

The green box indicates the set of memory addresses, eight bits long, where the number is stored. (The other bits, to the left and right of this box, might be used by the computer for other purposes, or simply unused.) What actual number does this sequence of eight bits represent? It depends: there are two types of formats for storing numbers, called **integers** and **floating-point numbers**.

3.2.1 Integers

In integer format, one of the bits is used to denote the sign of the number (positive or negative), and the remaining bits specify an integer in a binary representation. Because only a fixed number of bits is available, only a finite range of integers can be represented. On modern 64-bit computers, integers are typically stored using 64 bits, so only 2^{64} distinct integers can be represented. The minimum and maximum integers are

$$n_{\min} = -2^{63} \quad = -9223372036854775808 \tag{3.2.1}$$

$$n_{\max} = 2^{63} - 1 = +9223372036854775807 \tag{3.2.2}$$

Python is somewhat able to conceal this limitation through a "variable-width integer" feature. If the numbers you're dealing with exceed the above limits, it can convert the numbers to a different format, in which the number of bits can be arbitrarily larger than 64 bits. However, these variable-width integers have various limitations. For one thing, they cannot be stored in a Scipy array meant for storing standard fixed-width integers:

```
>>> n = array([-2**63])                 # Create an integer array.
>>> n                                   # Check the array's contents.
array([-9223372036854775808])
>>> n.dtype                             # Check the array's data type: it should stor
dtype('int64')
>>> m = -2**63 - 1                      # If we specify an integer that's too negativ
>>> m
-9223372036854775809
>>> n[0] = m                            # What if we try to store it in a 64-bit inte
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: Python int too large to convert to C long
```

Moreover, performing arithmetic operations on variable-width integers is much slower than standard fixed-width arithmetic. Therefore, it's generally best to avoid dealing with integers that are too large or too small.

3.2.2 Floating-Point Numbers

Floating-point numbers (sometimes called **floats**) are a format for approximately representing "real numbers"—numbers that are not necessarily integers. Each number is broken up into three components: the "sign", "fraction", and "exponent". For example, the charge of an electron can be represented as

$$q \approx -1.60217657 \times 10^{-19} = \Big(-\Big)\Big(160217657\Big) \times 10^{-27} \tag{3.2.3}$$

This number can thus be stored as one bit (the sign $-$), and two integers (the fraction $160217646$and the exponent -27). The two integers are stored in fixed-width format, so the overall floating-point number also has fixed width (64 bits in modern computers). The actual implementation details of floating-point numbers differs from the above example in a few ways—notably, the numbers are represented using powers of 2 rather than powers of 10—but this is the basic idea.

In Python code, floating-point numbers are specified in decimal notation (e.g. `0.000001`) or exponential notation (e.g. `1e-6`). If you want to represent an integer in floating-point format, give it a trailing decimal point, like this: `3.` (or, equivalently, `3.0`). If you perform arithmetic between an integer and a floating-point number, like `2 + 3.0` , the result is a floating-point number. And if you divide two integers, the result is a floating-point number:

```
>>> a, b = 6, 3   # a and b are both integers
>>> a/b           # a/b is in floating-point format
2.0
```

Note

In Python 2, dividing two integers yielded a rounded integer. This is a frequent source of bugs, so be aware of this behavior if you ever use Python 2.

Numerical Imprecision

Because floating-point numbers use a finite number of bits, the vast majority of real numbers cannot be *exactly* represented in floating-point format. They can only be approximated. This gives rise to interesting quirks, like this:

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

The reason is that the decimal number 0.1 does not correspond exactly to a floating-point representation, so when you tell the computer to create a number "0.1", it instead approximates that number using a floating-point number whose decimal value is $0.100000000000000055511512...$

Because of this lack of precision, you should not compare floating-point numbers using Python's equality operator `==` :

```
>>> x = 0.1 + 0.1 + 0.1
>>> y = 0.3
>>> x == y
False
```

Instead of using `==` , you can compare `x` and `y` by checking if they're closer than a certain amount:

```
>>> epsilon = 1e-6
>>> abs(x-y) < epsilon
True
```

The "density" of real numbers represented exactly floating point numbers decreases exponentially with the magnitude of the number. Hence, large floating-point numbers are less precise than small ones. For this reason, numerical algorithms (such as Gaussian elimination) often try to avoid multiplying by very large numbers, or dividing by very small numbers.

Special Values

Like integers, floating-point numbers have a maximum and minimum number that can be represented (this is unavoidable, since they have only a finite number of bits). Any number above the maximum is assigned a special value, `inf` (infinity):

```
>>> 1e308
1e+308
>>> 1e309
inf
```

Similarly, any number below the floating-point minimum is represented by `-inf`. There is another special value called `nan` (not-a-number), which represents the results of calculations which don't make sense, like "infinity minus infinity":

```
>>> x = 1e310
>>> x
inf
>>> y = x - x
>>> y
nan
```

If you ever need to check if a number is `inf`, you can use Scipy's `isinf` function. You can check for `nan` using Scipy's `isnan` function:

```
>>> isinf(x)
True
>>> isnan(y)
True
>>> isnan(x)
False
```

3.2.3 Complex Numbers

Complex numbers are not a fundamental data type. Python implements each complex number as a composite of two floating-point numbers (the real and imaginary parts). You can specify a complex number using the notation `X+Yj`:

```
>>> z = 2+1j
>>> z
(2+1j)
```

Python's arithmetic operations can handle arithmetic operations on complex numbers:

```
>>> u = 3.4-1.2j
>>> z * u
(8+1j)
>>> z/u
(0.4307692307692308+0.44615384615384623j)
```

You can retrieve the real and imaginary parts of a complex number using the `.real` and `.imag` slots:

```
>>> z.real
2.0
>>> z.imag
1.0
```

Alternatively, you can use Scipy's `real` and `imag` functions (which also work on arrays). Similarly, the `absolute` (or `abs`) and `angle` functions return the magnitude and argument of a complex number:

```
>>> z = 2+1j
>>> real(z)
array(2.0)
>>> imag(z)
array(1.0)
>>> absolute(z)
2.2360679774997898
>>> angle(z)
0.46364760900080609
```

3.3: Arrays

We will often have to deal with collections of several numbers, which requires organizing them into **data structures**. One of the data structures that we will use most frequently is the **array**, which is a fixed-size linear sequence of numbers. We have already discussed the basic usage of Scipy arrays in the previous article.

The memory layout of an array is shown schematically in Fig. 3.3.1. It consists of two separate regions of memory:

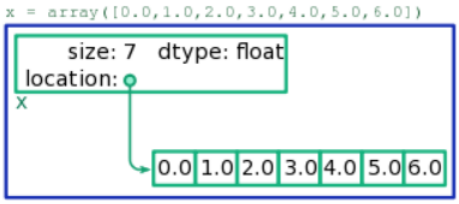

Figure 3.3.1: Schematic of how an array is laid out in memory. The book-keeping block (upper left box) records the array size, the address of the storage blocks (indicated by an arrow), etc. The storage blocks (lower right boxes) contain the array contents, in sequential order.

1. One region, which we call the **book-keeping block**, stores summary information about the array, including (i) the total number of elements, (ii) the memory address where the array contents are stored (specifically, the address of element 0), and (iii) the type of numbers stored in the array. The first two pieces of information are recorded in the form of integers, while the last piece is recorded in some other format that we don't need to worry about (it's managed by Python).
2. The second region, which we call the **data block**, stores the actual contents of the array, laid out sequentially. For example, for an array containing seven 64-bit integers, this block will consist of $7 \times 64 = 448$ bits of memory, storing the integers one after the other.

The book-keeping block and the data block aren't necessarily kept next to each other in memory. When a piece of Python code acts upon an array `x`, the information in the array's book-keeping block is used to locate the data block, and then access/alter its data as necessary.

3.3.1 Basic Array Operations

Let's take detailed look at what happens when we read or write an individual element of an array, say `x[2]`: the third element (index 2) stored in the array `x`.

From the array name, `x`, Python knows the address of the relevant book-keeping block (this is handled internally by Python, and takes negligible time). The book-keeping block records the address of element 0, i.e. the start of the data block. Because we want index 2 of the array, the processor jumps to the memory address that is 2 blocks past the recorded address. Since the data block is laid out sequentially, that is precisely the address where the number `x[2]` is stored. This number can now be read or overwritten, as desired by the Python code.

Under this scheme, the reading/writing of individual array elements is independent of the size of the array. Accessing an element in a size-1 array takes the same time as accessing an element in a size-100000 array. This is because the memory is random-access—the processor can jump to any address in memory once you tell it where to go. The memory layout of an array is designed so that one can always work out the relevant address in a single step.

We describe the speed of this operation using **big-O notation**. If N is the size of the array, reading/writing individual array elements is said to take $O(1)$ time, or "order-1 time" (i.e., independent of N). By contrast, a statement like

```
x.fill(3.3)
```

takes $O(N)$ time, i.e. time proportional to the array size N. That's because the `fill` method assign values to each of the N elements of the array. Similarly, the statement

```
x += 1.0
```

takes $O(N)$ time. This `+=` operation adds `1.0` to each of the elements of the array, which requires N arithmetic operations.

3.3.2 Array Data Type

We have noted that the book-keeping block of each array records the type of number, or **data type**, kept in the storage blocks. Thus, each individual array is able to store only one type of number. When you create an array with the `array` function, Scipy infers the data type based on the specified array contents. For example, if the input contains only integers, an integer array is created; if you then try to store a floating-point number, it will be rounded down to an integer:

```
>>> a = array([1,2,3,4])
>>> a[1] = 3.14159
>>> a
array([1, 3, 3, 4])
```

In the above situation, if our intention was to create an array of floating point numbers, that can be done by giving the `array` function an input containing at least one floating-point number. For example,

```
>>> a = array([1,2,3,4.])
>>> a[1] = 3.14159
>>> a
array([ 1.     ,  3.14159,  3.     ,  4.     ])
```

Alternatively, the `array` function accepts a parameter named `dtype`, which can be used to specify the data type directly:

```
>>> a = array([1,2,3,4], dtype=float)
>>> a[1] = 3.14159
>>> a
array([ 1.     ,  3.14159,  3.     ,  4.     ])
```

The `dtype` parameter accepts several possible values, but most of time you will choose one of these three:

- `float`
- `complex`
- `integer`

The common functions for creating new arrays, `zeros` `ones`, and `linspace`, create arrays with the `float` data type by default. They also accept `dtype` parameters, in case you want a different data type. For example:

```
>>> a = zeros(4, dtype=complex)
>>> a[1] = 2.5+1j
>>> a
array([ 0.0+0.j,  2.5+1.j,  0.0+0.j,  0.0+0.j])
```

3.3.3 Vectorization

We have previously discussed the code `x += 1.0`, which adds `1.0` to every element on the array `x`. It has runtime $O(N)$, where N is the array length. We could also have done the same thing by looping over the array, as follows:

```
for n in range(len(x)):
    x[n] += 1.0
```

This, too, has $O(N)$ runtime. But it is not a good way to do the job, for two reasons. Firstly, it's obviously much more cumbersome to write. Secondly, and more importantly, it is much more inefficient, because it involves more "high-level" Python operations. To

run this code, Python has to create an index variable `n`, increment that index variable N times, and increment `x[n]` for each separate value of `n`.

By contrast, when you write `x += 1.0`, Python uses "low-level" code to increment each element in the array, which does not require introducing and managing any "high-level" Python objects. The practice of using array operations, instead of performing explicit loops over an array, is called **vectorization**. You should always strive to vectorize your code; it is generally good programming practice, and leads to extreme performance gains for large array sizes.

Vectorization does not change the runtime scaling of the operation. The vectorized code `x += 1.0`, and the explicit loop, both run in $O(N)$ time. What changes is the *coefficient* of the scaling: the runtime has the form T, and the value of the coefficient a is much smaller for vectorized code.

Here is another example of vectorization. Suppose we have a variable `y` whose value is a number, and an array `x` containing a collection of numbers; we want to find the element of `x` closest to `y`. Here is non-vectorized code for doing this:

```
idx, distance = 0, abs(x[0] - y)
for n in range(1, len(x)):
    new_dist = abs(x[n] - y)
    if new_dist < distance:
        idx, distance = n, new_dist

z = x[idx]
```

The vectorized approach would simply make use of the `argmin` function:

```
idx = argmin(abs(x - y))
z = x[idx]
```

The way this works is to create a new array, whose values are the distances between each element of `x` and the target number `y`; then, `argmin` searches for the array index corresponding to the smallest element (which is also the index of the element of `x` closest to `y`). We could write this code even more compactly as

```
z = x[argmin(abs(x - y))]
```

3.3.4 Array Slicing

We have emphasized that an array is laid out in memory in two pieces: a book-keeping block, and a sequence of storage blocks containing the elements of the array. Sometimes, it is possible for two arrays to share storage blocks. For instance, this happens when you perform array slicing:

```
>>> x = array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
>>> y = x[2:5]
>>> y
array([ 2.,  3.,  4.])
```

The statement `y = x[2:5]` creates an array named `y`, containing a subset of the elements of `x` (i.e., the elements at indices 2, 3, and 4). However, Python does *not* accomplish this by copying the affected elements of `x` into a new array with new storage blocks. Instead, it creates a new book-keeping block for `y`, and points it towards the existing storage blocks of `x`:

Figure 3.3.2: Two arrays, `x` and `y`, sharing the same storage blocks.

Because the storage blocks are shared between two arrays, if we change an element in `x`, that effectively changes the contents of `y` as well:

```
>>> x[3] = 9.
>>> y
array([ 2.,  9.,  4.])
```

(The situation is similar if you specify a "step" during slicing, like `y = x[2:5:2]`. What happens in that case is that the data block keeps track of the step size, and Python can use this to figure out exactly which address to jump for accessing any given element.)

The neat thing about this method of sharing storage blocks is that slicing is an $O(1)$ operation, independent of the array size. Python does not need to do any copying on the stored elements; it merely needs to create a new book-keeping block. Therefore, slicing is a very "cheap" and efficient operation.

The downside is that it can lead to strange bugs. For example, this is a common mistake:

```
>>> x = y = linspace(0, 1, 100)
```

The above statement creates two arrays, `x` and `y`, pointing to the *same* storage blocks. This is almost definitely not what we intend! The correct way is to write two separate array initialization statements.

Whenever you intend to copy an array and change its contents freely without affecting the original array, you must remember to use the `copy` function:

```
>>> x = array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
>>> y = copy(x[2:5])
>>> x[3] = 9.
>>> y
array([ 2.,  3.,  4.])
```

In the above example, the statement `y = copy(x[2:5])` explicitly copies out the storage blocks of `x`. Therefore, when we change the contents of `x`, the contents of `y` are unaffected.

Do not call `copy` too liberally! It is an $O(N)$ operation, so unnecessary copying hurts performance. In particular, the basic arithmetic operations don't affect the contents of arrays, so it is always safe to write

```
>>> y = x + 4
```

rather than `y = copy(x) + 4`.

3.4: Exercises

Exercise 3.4.1

Traditionally, computers keep track of the time/date using a format known as Unix time, which counts the number of seconds that have elapsed since 00:00:00 UTC on Thursday, 1 January 1970. But there's a problem if we track Unix time using a fixed-width integer, since that has a maximum value. Beyond this date, the Unix time counter will roll-over, wreaking havoc on computer systems. Calculate the roll-over date for:

1. Ordinary (signed) 32-bit integers
2. *Unsigned* 32-bit integers, which do not reserve a bit for the sign (and thus store only non-negative numbers).
3. Signed 64-bit integers
4. Unsigned 64-bit integers

Exercise 3.4.2

Find the runtime of each of the following Python code samples (e.g. $O(1)$ or $O(N)$). Assume that the arrays `x` and `y` are of size N:

a. `z = x + y`
b. `x[5] = x[4]`
c. `z = conj(x)`
d. `z = angle(x)`
e. `x = x[::-1]` (this reverses the order of elements).

Exercise 3.4.3

Write a Python function `uniquify_floats(x, epsilon)`, which accepts a list (or array) of floats `x`, and deletes all "duplicate" elements that are separated from another element by a distance of less than `epsilon`. The return value should be a list (or array) of floats that differ from each other by at least `eps`.

Exercise 3.4.4

(Hard) Suppose a floating-point representation uses one sign bit, N fraction bits, and M exponent bits. Find the density of real numbers which can be represented exactly by a floating-point number. Hence, show that floating-point precision decreases exponentially with the magnitude of the number.

CHAPTER OVERVIEW

4: Numerical Linear Algebra

Much of scientific programming involves handling numerical linear algebra. This is because a huge number of numerical problems which occur in science can be reduced to linear algebra problems. It is very important to know how to formulate these linear algebra problems, and how to solve them numerically.

4.1: Array Representations of Vectors, Matrices, and Tensors

Thus far, we have discussed simple "one-dimensional" (1D) arrays, which are linear sequences of numbers. In linear algebra terms, 1D arrays represent **vectors**. The array length corresponds to the "vector dimension" (e.g., a 1D array of length 3 corresponds to a 3-vector). In accordance with Scipy terminology, we will use the work "dimension" to refer to the dimensionality of the array (called the **rank** in linear algebra), and not the vector dimension.

You are probably familiar with the fact that vectors can be represented using **index notation**, which is pretty similar to Python's notation for addressing 1D arrays. Consider a length-d vector

$$\vec{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{d-1} \end{bmatrix}. \tag{4.1.1}$$

The jth element can be written as

$$x_j \quad \leftrightarrow \quad \texttt{x[j]}, \tag{4.1.2}$$

where $j = 0, 1, \ldots, d-1$. The notation on the left is mathematical index notation, and the notation on the right is Python's array notation. Note that we are using 0-based indexing, so that the first element has index 0 and the last element has index $d-1$.

A **matrix** is a collection of numbers organized using two indices, rather than a single index like a vector. Under 0-based indexing, the elements of an $m \times n$ matrix are:

$$\mathbf{M} = \begin{bmatrix} M_{00} & M_{01} & \cdots & M_{0,n-1} \\ M_{10} & M_{11} & \cdots & M_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ M_{m-1,0} & M_{m-1,1} & \cdots & M_{m-1,n-1} \end{bmatrix}. \tag{4.1.3}$$

More generally, numbers that are organized using multiple indices are collectively referred to as **tensors**. Tensors can have more than two indices. For example, vector cross products are computed using the Levi-Civita tensor ε, which has three indices:

$$\left(\vec{A} \times \vec{B}\right)_i = \sum_{jk} \varepsilon_{ijk} A_j B_k. \tag{4.1.4}$$

4.1.1 Multi-Dimensional Arrays

In Python, tensors are represented by **multi-dimensional arrays**, which are similar to 1D arrays except that they are addressed using more than one index. For example, matrices are represented by 2D arrays, and the (i, j)th component of an $m \times n$ matrix is written in Python notation as follows:

$$M_{ij} \quad \leftrightarrow \quad \texttt{M[i,j]} \qquad \text{for } \; i = 0, \ldots, m-1, \;\; j = 0, \ldots, n-1. \tag{4.1.5}$$

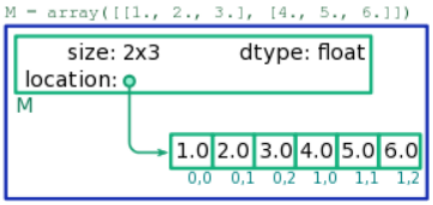

Figure 4.1.1: Memory model of a 2D array.

The way multi-dimensional arrays are laid out in memory is very similar to the memory layout of 1D arrays. There is a book-keeping block, which is associated with the array name, and which stores information about the array size (including the number of indices and the size of each index), as well as the memory location of the array contents. The elements lie in a sequence of storage blocks, in a specific order (depending on the array size). This arrangement is shown schematically in Fig. 4.1.1.

When Python needs to access any element of of a multi-dimensional array, it knows exactly which memory location the element is stored in. The location can be worked out from the size of the multi-dimensional array, and the memory location of the first element. In Fig. 4.1.1, for example, `M` is a 2×3 array, containing 6 storage blocks laid out in a specific sequence. If we need to access `M[1,1]`, Python knows that it needs to jump to the storage block four blocks down from the $(0, 0)$ block. Hence, reading/writing the elements of a multi-dimensional array is an $O(1)$ operation, just like for 1D arrays.

In the following subsections, we will describe how multi-dimensional arrays can be created and manipulated in Python code.

Note

There is also a special Scipy class called `matrix` which can be used to represent matrices. *Don't use this.* It's a layer on top of Scipy's multi-dimensional array facilities, mostly intended as a crutch for programmers transitioning from Matlab. Arrays are better to use, and more consistent with the rest of Scipy.

4.1.2 Creating Multi-Dimensional Arrays

You can create a 2D array with specific elements using the `array` command, with an input consisting of a list of lists:

```
>>> x = array([[1., 2., 3.], [4., 5., 6.]])
>>> x
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

The above code creates a 2D array (i.e. a matrix) named `x`. It is a 2×3 array, containing the elements $x_{00} = 1$, $x_{01} = 2$, $x_{02} = 3$, etc. Similarly, you can create a 3D array by supplying an input consisting of a list of lists of lists; and so forth.

It is more common, however, to create multi-dimensional arrays using `ones` or `zeros`. These functions return arrays whose elements are all initialized to 0.0 and 1.0, respectively. (You can then assign values to the elements as desired.) To do this, instead of specifying a number as the input (which would create a 1D array of that size), you should specify a tuple as the input. For example,

```
>>> x = zeros((2,3))
>>> x
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
>>> y = ones((3,2))
>>> y
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

There are many more ways to create multi-dimensional arrays, which we'll discuss when needed.

4.1.3 Basic Array Operations

To check on the dimension of an array, consult its `ndim` slot:

```
>>> x = zeros((5,4))
>>> x.ndim
2
```

To determine the exact shape of the array, use the `shape` slot (the shape is stored in the form of a tuple):

```
>>> x.shape
(3, 5)
```

To access the elements of a multi-dimensional array, use square-bracket notation: `M[2,3]`, `T[0,4,2]`, etc. Just remember that each component is zero-indexed.

Multi-dimensional arrays can be sliced, similar to 1D arrays. There is one important feature of multi-dimensional slicing: if you specify a single value as one of the indices, the slice results in an array of smaller dimensionality. For example:

```
>>> x = array([[1., 2., 3.], [4., 5., 6.]])
>>> x[:,0]
array([ 1.,  4.])
```

In the above code, `x` is a 2D array of size 2×3. The slice $x[:, 0]$ specifies the value 0 for index 1, so the result is a 1D array containing the elements $[x_{00}, x_{10}]$.

If you don't specify all the indices of a multi-dimensional array, the omitted indices implicitly included, and run over their entire range. For example, for the above `x` array,

```
>>> x[1]
array([ 4.,  5.,  6.])
```

This is also equivalent to `x[1,:]`.

4.1.4 Arithmetic Operations

The basic arithmetic operations can all be performed on multi-dimensional arrays, and act on the arrays element-by-element. For example,

```
>>> x = ones((2,3))
>>> y = ones((2,3))
>>> z = x + y
>>> z
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
```

You can think of this in terms of index notation:

$$z_{ij} = x_{ij} + y_{ij}. \tag{4.1.6}$$

What is the runtime for performing such arithmetic operations on multi-dimensional arrays? With a bit of thinking, we can convince ourselves that the runtime scales linearly with the number of elements in the multi-dimensional array, because the arithmetic operation is performed on each individual index. For example, the runtime for adding a pair of $M \times N$ matrices scales as $(O(MN)$.

Note

The multiplication operator `*` also acts element-by-element. It does *not* refer to matrix multiplication!

For example,

```
>>> x = ones((2,3))
>>> y = ones((2,3))
>>> z = x * y
```

```
>>> z
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

In index notation, we can think of the `*` operator as doing this:

$$z_{ij} = x_{ij} y_{ij}. \tag{4.1.7}$$

By contrast, matrix multiplication is $z_{ij} = \sum_k x_{ik} y_{kj}$. We'll see how this is accomplished in the next subsection.

4.1.5 The Dot Operation

The most commonly-used function for array multiplication is the `dot` function, which takes two array inputs `x` and `y` and returns their "dot product". It constructs a product by summing over the *last* index of array `x`, and over the *next-to-last* index of array `y` (or over its last index, if `y` is a 1D array). This may sound like a complicated rule, but you should be able to convince yourself that it corresponds to the appropriate type of multiplication operation for the most common cases encountered in linear algebra:

- If `x` and `y` are both 1D arrays (vectors), then `dot` corresponds to the usual dot product between two vectors:

$$\texttt{z = dot(x,y)} \quad \leftrightarrow \quad z = \sum_k x_k \, y_k \tag{4.1.8}$$

- If `x` is a 2D array and `y` is a 1D array, then `dot` corresponds to right-multiplying a matrix by a vector:

$$\texttt{z = dot(x,y)} \quad \leftrightarrow \quad z_i = \sum_k x_{ik} \, y_k \tag{4.1.9}$$

- If `x` is a 1D array and `y` is a 2D array, then `dot` corresponds to left-multiplication:

$$\texttt{z = dot(x,y)} \quad \leftrightarrow \quad z_i = \sum_k x_k \, y_{ki} \tag{4.1.10}$$

- If `x` and `y` are both 2D arrays, `dot` corresponds to matrix multiplication:

$$\texttt{z = dot(x,y)} \quad \leftrightarrow \quad z_{ij} = \sum_k x_{ik} \, y_{kj} \tag{4.1.11}$$

- The rule applies to higher-dimensional arrays as well. For example, two rank-3 tensors are multiplied together in this way:

$$\texttt{z = dot(x,y)} \quad \leftrightarrow \quad z_{ijpq} = \sum_k x_{ijk} \, y_{pkq} \tag{4.1.12}$$

Should you need to perform more general products than what the `dot` function provides, you can use the `tensordot` function. This takes two array inputs, `x` and `y`, and a tuple of two integers specifying which components of `x` and `y` to sum over. For example, if `x` and `y` are 2D arrays,

$$\texttt{z = tensordot(x, y, (0,1))} \quad \leftrightarrow \quad z_{ij} = \sum_k x_{ki} \, y_{jk} \tag{4.1.13}$$

What is the runtime for `dot` and `tensordot`? Consider a simple case: matrix multiplication of an $M \times N$ matrix with an $N \times P$ matrix. In index notation, this has the form

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \, B_{kj}, \quad \text{for } i \in \{0, \ldots, M-1\}, \;\; j \in \{0, \ldots, P-1\}. \tag{4.1.14}$$

The resulting matrix has a total of $(M \times P)$ indices to be computed. Each of these calculations requires a sum involving $O(N)$ arithmetic operations. Hence, the total runtime scales as $O(MNP)$. By similar reasoning, we can figure out the runtime scaling for any tensor product between two tensors: it is the product of the sizes of the unsummed indices, times the size of the summed index. For example, for a `tensordot` product between an $M \times N \times P$ tensor and a $Q \times S \times P$ tensor, summing over the last index of each tensor, the runtime would scale as $O(MNPQS)$.

4.2: Linear Equations

In physics, we are often called upon to solve **linear equations** of the form

$$\mathbf{A}\vec{x} = \vec{b}, \tag{4.2.1}$$

where $\mathbf{A}$ is some $N \times N$ matrix, and both $\vec{x}$ and $\vec{b}$ are vectors for length N. Given $\mathbf{A}$ and $\vec{b}$, the goal is to solve for $\vec{x}$.

It's an important and useful skill to recognize linear systems of equations when they arise in physics problems. Such equations can arise in many diverse contexts; we will give a couple of simple examples below.

Example 4.2.1

Suppose there is a set of N electrically charged point particles at positions $\{\vec{R}_0, \vec{R}_1, \ldots, \vec{R}_{N-1}\}$. We do not know the value of the electric charges, but we able to measure the electric potential at any point $\vec{r}$. The electric potential is given by

$$\phi(\vec{r}) = \sum_{j=0}^{N-1} \frac{q_j}{|\vec{r} - \vec{R}_j|}. \tag{4.2.2}$$

If we measure the potential at N positions, $\{\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_{N-1}\}$, how can the charges $\{q_0, \ldots, q_{N-1}\}$ be deduced?

Solution

To do this, let us write the equation for the electric potential at point $\vec{r}_i$ as:

$$\phi(\vec{r}_i) = \sum_{j=0}^{N-1} \left[\frac{1}{|\vec{r}_i - \vec{R}_j|} \right] q_j. \tag{4.2.3}$$

This has the form $\mathbf{A}\vec{x} = \vec{b}$, where $\mathbf{A}_{ij} \equiv \frac{1}{|\vec{r}_i - \vec{R}_j|}$, $\vec{b}_i \equiv \phi(\vec{r}_i)$, and the unknowns are $\vec{x}_j = q_j$.

Example 4.2.2

Linear systems of equations commonly appear in circuit theory. For example, consider the following parallel circuit of N power supplies and resistances:

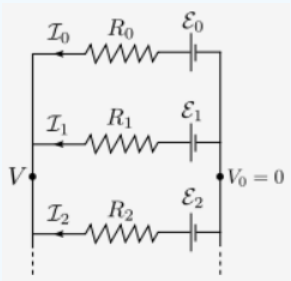

Figure 4.2.1

Assume the voltage on the right-hand side of the circuit is $V_0 = 0$. Given the resistances $\{R_0, \ldots, R_{N-1}\}$ and the EMFs $\{\mathcal{E}_0, \ldots, \mathcal{E}_{N-1}\}$, how do we find the left-hand voltage V and the currents $\{\mathcal{I}_0, \ldots, \mathcal{I}_{N-1}\}$?

Solution

We follow the usual laws of circuit theory. Each branch of the parallel circuit obeys Ohm's law,

$$\mathcal{I}_j R_j + V = \mathcal{E}_j. \tag{4.2.4}$$

Furthermore, the currents obey Kirchoff's law (conservation of current), so

$$\sum_{j=0}^{N-1} \mathcal{I}_j = 0. \tag{4.2.5}$$

We can combine these $N+1$ equations into a matrix equation of the form $\mathcal{A}\,\vec{x} = \vec{b}$

$$\begin{bmatrix} R_0 & 0 & \cdots & 0 & 1 \\ 0 & R_1 & \cdots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & R_{N-1} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathcal{I}_0 \\ \mathcal{I}_1 \\ \vdots \\ \mathcal{I}_{N-1} \\ V \end{bmatrix} = \begin{bmatrix} \mathcal{E}_0 \\ \mathcal{E}_1 \\ \vdots \\ \mathcal{E}_{N-1} \\ 0 \end{bmatrix} \tag{4.2.6}$$

Here, the unknown vector $\vec{x}$ consists of the N currents passing through the branches of the circuit, and the potential V.

4.2.1 Direct Solution

Faced with a system of linear equations, one's first instinct is usually to solve for $\vec{x}$ by inverting the matrix $\mathbf{A}$:

$$\mathbf{A}\vec{x} = \vec{b} \quad \Rightarrow \quad \vec{x} = \mathbf{A}^{-1}\,\vec{b}. \tag{4.2.7}$$

Don't do this. It is mathematically correct, but numerically inefficient. As we'll see, computing the matrix inverse $\mathbf{A}^{-1}$, and then right-multiplying by $\vec{b}$, involves more steps than simply solving the equation directly

To solve a system of linear equations, use the `solve` function from the `scipy.linalg` module. (You will need to import `scipy.linalg` explicitly, because it is a submodule of `scipy` and does not get imported by our usual `from scipy import *` statement.) Here is an example:

```
>>> A = array([[1., 2., 3.], [2., 4., 0.], [1., 3., 9.]])
>>> b = array([6., 6., 9.])
>>>
>>> import scipy.linalg as lin
>>> x = lin.solve(A, b)
>>> x
array([ 9., -3.,  1.])
```

We can verify that this is indeed the solution:

```
>>> dot(A, x)              # This should equal b.
array([ 6.,  6.,  9.])
```

The direct solver uses an algorithm known as Gaussian elimination, which we'll discuss in the next article. The runtime of Gaussian elimination is $O(N^3)$, where N is the size of the linear algebra problem.

The reason we avoid solving linear equations by inverting the matrix $\mathbf{A}$ is that the matrix inverse is itself calculated using the Gaussian elimination algorithm! If you are going to use Gaussian elimination anyway, it is far better to apply the algorithm directly on the desired $\mathbf{A}$ and b. Solving by calculating $\mathbf{A}^{-1}$ involves about twice as many computational steps.

4.3: Exercises

Exercise 4.3.1

Write Python code to construct a 3D array of size $3 \times 3 \times 3$ corresponding to the Levi-Civita tensor,

$$\varepsilon_{ijk} = \begin{cases} +1 & \text{if } (i,j,k) \text{ is } (1,2,3), (2,3,1) \text{ or } (3,1,2), \\ -1 & \text{if } (i,j,k) \text{ is } (3,2,1), (1,3,2) \text{ or } (2,1,3), \\ 0 & \text{if } i=j \text{ or } j=k \text{ or } k=i \end{cases} \tag{4.3.1}$$

Then, using the `tensordot` function, verify the identity $\sum_i \varepsilon_{ijk}\varepsilon_{imn} = \delta_{jm}\delta_{kn} - \delta_{jn}\delta_{km}$.

CHAPTER OVERVIEW

5: Gaussian Elimination

This article discusses the **Gaussian elimination** algorithm, one of the most fundamental and important numerical algorithms of all time. It is used to solve linear equations of the form

$$\mathbf{A}\vec{x} = \vec{b}, \tag{5.1}$$

where $\mathbf{A}$ is a known $N \times N$ matrix, $\vec{b}$ is a known vector of length N, and $\vec{x}$ is an unknown vector of length N. The goal is to find $\vec{x}$. The Gaussian elimination algorithm is implemented by Scipy's `scipy.linalg.solve` function.

5.1: The Basic Algorithm

The best way to understand how Gaussian elimination works is to work through a concrete example. Consider the following $N = 3$ problem:

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \\ 2 & 6 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 4 \end{bmatrix}. \tag{5.1.1}$$

The Gaussian elimination algorithm consists of two distinct phases: **row reduction** and **back-substitution**.

5.1.1 Row Reduction

In the row reduction phase of the algorithm, we manipulate the matrix equation so that the matrix becomes upper triangular (i.e., all the entries below the diagonal are zero). To achieve this, we note that we can subtract one row from another row, without altering the solution. In fact, we can subtract any *multiple* of a row.

We will eliminate (zero out) the elements below the diagonal in a specific order: from *top to bottom* along each column, then from *left to right* for successive columns. For our 3×3 example, the elements that we intend to eliminate, and the order in which we will eliminate them, are indicated by the colored numbers 0, 1, and 2 in the following figure:

$$\begin{bmatrix} 1 & 2 & 3 \\ 3_0 & 2 & 2 \\ 2_1 & 6_2 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 4 \end{bmatrix}$$

Figure 5.1.1

The first matrix element we want to eliminate is at $(1, 0)$ (orange circle). To eliminate it, we subtract, from this row, a multiple of row 0. We will use a factor of $3/1 = 3$:

$$(3x_0 + 2x_1 + 2x_2) - (3/1)(1x_0 + 2x_1 + 3x_2) = 4 - (3/1)3 \tag{5.1.2}$$

The factor of 3 we used is determined as follows: we divide the matrix element at $(1, 0)$ (which is the one we intend to eliminate) by the element at $(0, 0)$ (which is the one along the diagonal in the same column). As a result, the term proportional to x_0 disappears, and we obtain the following modified linear equations, *which possess the same solution*:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0_0 & -4 & -7 \\ 2_1 & 6_2 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -5 \\ 4 \end{bmatrix}$$

Figure 5.1.2

(Note that we have changed the entry in the vector on the right-hand side as well, not just the matrix on the left-hand side!) Next, we eliminate the element at $(2, 0)$ (green circle). To do this, we subtract, from this row, a multiple of row 0. The factor to use is $2/1 = 2$, which is the element at $(2, 0)$ divided by the $(0, 0)$ (diagonal) element:

$$(2x_0 + 6x_1 + 2x_2) - (2/1)(1x_0 + 2x_1 + 3x_2) = 4 - (2/1)3 \tag{5.1.3}$$

The result is

$$\begin{bmatrix} 1 & 2 & 3 \\ 0_0 & -4 & -7 \\ 0_1 & 2_2 & -4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -5 \\ -2 \end{bmatrix}$$

Figure 5.1.3

Next, we eliminate the $(2, 1)$ element (blue circle). This element lies in column 1, so we eliminate it by subtracting a multiple of row 1. The factor to use is $2/(-4) = -0.5$, which is the $(2, 1)$ element divided by the $(1, 1)$ (diagonal) element:

$$(0x_0 + 2x_1 - 4x_2) - (2/(-4))(0x_0 - 4x_1 - 7x_2) = -5 - (2/(-4))(-2) \tag{5.1.4}$$

The result is

$$\begin{bmatrix} 1 & 2 & 3 \\ 0_0 & -4 & -7 \\ 0_1 & 0_2 & -7.5 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -5 \\ -4.5 \end{bmatrix}$$

Figure 5.1.4

We have now completed the row reduction phase, since the matrix on the left-hand side is upper-triangular (i.e., all the entries below the diagonal have been set to zero).

5.1.2 Back-Substitution

In the back-substitution phase, we read off the solution from the bottom-most row to the top-most row. First, we examine the bottom row:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -4 & -7 \\ 0 & 0 & -7.5 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -5 \\ -4.5 \end{bmatrix}$$

Figure 5.1.5

Thanks to row reduction, all the matrix elements on this row are zero except for the last one. Hence, we can read off the solution

$$x_2 = (-4.5)/(-7.5) = 0.6. \tag{5.1.5}$$

Next, we look at the row above:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -4 & -7 \\ 0 & 0 & -7.5 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -5 \\ -4.5 \end{bmatrix}$$

Figure 5.1.6

This is an equation involving x_1 and x_2. But from the previous back-substitution step, we know x_2. Hence, we can solve for

$$x_1 = [-5 - (-7)(0.6)]/(-4) = 0.2. \tag{5.1.6}$$

Finally, we look at the row above:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -4 & -7 \\ 0 & 0 & -7.5 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -5 \\ -4.5 \end{bmatrix}$$

Figure 5.1.7

This involves all three variables x_0, x_1, and x_2. But we already know x_1 and x_2, so we can read off the solution for x_0. The final result is

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0.2 \\ 0.6 \end{bmatrix}. \tag{5.1.7}$$

5.1.3 Runtime

Let's summarize the components of the Gaussian elimination algorithm, and analyze how many steps each part takes:

Row reduction

•	Step forwards through the rows. For each row n,			N steps
	•	Perform pivoting (to be discussed below).		$N - n + 1 \sim N$ steps
	•	Step forwards through the rows larger than n. For each such row m,		$N - n \sim N$ steps
		•	Subtract (A'_{mn}/A'_{nn}) times row n from the row m (where $\mathbf{A}'$ is the current matrix). This eliminates the matrix element at (m, n).	$O(N)$ arithmetic operations
Back-substitution				
•	Step backwards through the rows. For each row n,			N steps
	•	Substitute in the solutions x_m for $m > n$ (which are already found). Hence, find x_n.		$N - n \sim O(N)$ arithmetic operations

(The "pivoting" procedure hasn't been discussed yet; we'll do that in a later section.)

We conclude that the runtime of the row reduction phase scales as $O(N^3)$, and the runtime of the back-substitution phase scales as $O(N^2)$. The algorithm's overall runtime therefore scales as $O(N^3)$.

5.2: Matrix Generalization

We can generalize the Gaussian elimination algorithm described in the previous section, to solve matrix problems of the form

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}, \tag{5.2.1}$$

where $\mathbf{x}$ and $\mathbf{b}$ are $N \times M$ matrices, not merely vectors. An example, for $M = 2$, is

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \\ 2 & 6 & 2 \end{bmatrix} \begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \\ x_{20} & x_{21} \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 4 & 8 \\ 4 & 2 \end{bmatrix}. \tag{5.2.2}$$

It can get a bit tedious to keep writing out the x elements in the system of equations, particularly when $\mathbf{x}$ becomes a matrix. For this reason, we switch to a notation known as the **augmented matrix**:

$$\left[\begin{array}{ccc|cc} 1 & 2 & 3 & 3 & 6 \\ 3 & 2 & 2 & 4 & 8 \\ 2 & 6 & 2 & 4 & 2 \end{array}\right]. \tag{5.2.3}$$

Here, the entries to the left of the vertical separator denote the left-hand side of the system of equations, and the entries to the right of the separator denote the right-hand side of the system of equations.

The Gaussian elimination algorithm can now be performed directly on the augmented matrix. We will walk through the steps for the above example. First, row reduction:

- Eliminate the element at $(1, 0)$:

$$\left[\begin{array}{ccc|cc} 1 & 2 & 3 & 3 & 6 \\ 0 & -4 & -7 & -5 & -10 \\ 2 & 6 & 2 & 4 & 2 \end{array}\right] \tag{5.2.4}$$

- Eliminate the element at $(2, 0)$:

$$\left[\begin{array}{ccc|cc} 1 & 2 & 3 & 3 & 6 \\ 0 & -4 & -7 & -5 & -10 \\ 0 & 2 & -4 & -2 & -10 \end{array}\right] \tag{5.2.5}$$

- Eliminate the element at $(2, 1)$:

$$\left[\begin{array}{ccc|cc} 1 & 2 & 3 & 3 & 6 \\ 0 & -4 & -7 & -5 & -10 \\ 0 & 0 & -7.5 & -4.5 & -15 \end{array}\right] \tag{5.2.6}$$

The back-substitution step converts the left-hand portion of the augmented matrix to the identity matrix:

- Solve for row 2:

$$\left[\begin{array}{ccc|cc} 1 & 2 & 3 & 3 & 3 \\ 0 & -4 & -7 & -5 & -10 \\ 0 & 0 & 1 & 0.6 & 2 \end{array}\right] \tag{5.2.7}$$

- Solve for row 1:

$$\left[\begin{array}{ccc|cc} 1 & 2 & 3 & 3 & 3 \\ 0 & 1 & 0 & 0.2 & -1 \\ 0 & 0 & 1 & 0.6 & 2 \end{array}\right] \tag{5.2.8}$$

- Solve for row 0:

$$\left[\begin{array}{ccc|cc} 1 & 0 & 0 & 0.8 & 2 \\ 0 & 1 & 0 & 0.2 & -1 \\ 0 & 0 & 1 & 0.6 & 2 \end{array}\right] \tag{5.2.9}$$

After the algorithm finishes, the right-hand side of the augmented matrix contains the result for $\mathbf{x}$. Analyzing the runtime using the same reasoning as before, we find that the row reduction step scales as $O\Big(N^2(N+M)\Big)$, and the back-substitution step scales as $O\Big(N(N+M)\Big)$.

This matrix form of the Gaussian elimination algorithm is the standard method for computing matrix inverses. If $\mathbf{b}$ is the $N \times N$ identity matrix, then the solution $\mathbf{x}$ will be the inverse of $\mathbf{A}$. Thus, the runtime for calculating a matrix inverse scales as $O(N^3)$.

5.3: Pivoting

In our description of the Gaussian elimination algorithm so far, you may have noticed a problem. During the row reduction process, we have to multiply rows by a factor of A'_{mn}/A'_{nn} (where $\mathbf{A}'$ denotes the current matrix). If A'_{nn} happens to be zero, the factor blows up, and the algorithm fails.

To bypass this difficulty, we add an extra step to the row reduction procedure. As we step forward through row numbers $n = 0, 1, \cdots, N-1$, we do the following for each n:

- *Pivoting*: search through the matrix elements on and below the diagonal element at (n, n), and find the row n' with the largest value of $|A'_{n'n}|$. Then, swap row n and row n'.
- Continue with the rest of the algorithm, eliminating the A'_{mn} elements below the diagonal.

You should be able to convince yourself that (i) pivoting does not alter the solution, and (ii) it does not alter the runtime scaling of the row reduction phase, which remains $O(N^3)$.

Apart from preventing the algorithm from failing unnecessarily, pivoting improves its numerical stability. If A'_{nn} is non-zero but very small in magnitude, dividing by it will produce a very large result, which brings about a loss of floating-point numerical precision. Hence, it is advantageous to swap rows around to ensure that the magnitude of A'_{nn} is as large as possible.

When trying to pivot, it might happen that *all* the values of $|A'_{n'n}|$, on and below the diagonal, are zero (or close enough to zero within our floating-point tolerance). If this happens, it indicates that our original $\mathbf{A}$ matrix is singular, i.e., it has no inverse. Hence, the pivoting procedure has the additional benefit of helping us catch the cases where there is no valid solution to the system of equations; in such cases, the Gaussian elimination algorithm should abort.

5.3.1 Example

Let's work through an example of Gaussian elimination with pivoting, using the problem in the previous section:

$$\left[\begin{array}{ccc|cc} 1 & 2 & 3 & 3 & 6 \\ 3 & 2 & 2 & 4 & 8 \\ 2 & 6 & 2 & 4 & 2 \end{array}\right]. \tag{5.3.1}$$

The row reduction phase goes as follows:

- $(n = 0)$: Pivot, swapping row 0 and row 1:

$$\left[\begin{array}{ccc|cc} 3 & 2 & 2 & 4 & 8 \\ 1 & 2 & 3 & 3 & 6 \\ 2 & 6 & 2 & 4 & 2 \end{array}\right]. \tag{5.3.2}$$

- $(n = 0)$: Eliminate the element at $(1, 0)$:

$$\left[\begin{array}{ccc|cc} 3 & 2 & 2 & 4 & 8 \\ 0 & 4/3 & 7/3 & 5/3 & 10/3 \\ 2 & 6 & 2 & 4 & 2 \end{array}\right]. \tag{5.3.3}$$

- $(n = 0)$: Eliminate the element at $(2, 0)$:

$$\left[\begin{array}{ccc|cc} 3 & 2 & 2 & 4 & 8 \\ 0 & 4/3 & 7/3 & 5/3 & 10/3 \\ 0 & 14/3 & 2/3 & 4/3 & -10/3 \end{array}\right]. \tag{5.3.4}$$

- $(n = 1)$: Pivot, swapping row 1 and row 2:

$$\left[\begin{array}{ccc|cc} 3 & 2 & 2 & 4 & 8 \\ 0 & 14/3 & 2/3 & 4/3 & -10/3 \\ 0 & 4/3 & 7/3 & 5/3 & 10/3 \end{array}\right]. \tag{5.3.5}$$

- ($n = 1$): Eliminate the element at $(2, 1)$:

$$\left[\begin{array}{ccc|cc} 3 & 2 & 2 & 4 & 8 \\ 0 & 14/3 & 2/3 & 4/3 & -10/3 \\ 0 & 0 & 15/7 & 9/7 & 30/7 \end{array}\right]. \tag{5.3.6}$$

The back-substitution phase then proceeds as usual. You can check that it gives the same results we obtained before.

5.4: LU Decomposition

A variant of the Gaussian elimination algorithm can be used to compute the **LU decomposition** of a matrix. This procedure was invented by Alan Turing, the British mathematician considered the "father of computer science". The LU decomposition of a square matrix $\mathbf{A}$ consists of a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$, such that

$$\mathbf{A} = \mathbf{L}\,\mathbf{U}. \tag{5.4.1}$$

In certain special circumstances, LU decompositions provide a very efficient method for solving linear equations. Suppose that we have to solve a set of linear equations $\mathbf{A}\vec{x} = \vec{b}$ many times, using the *same* $\mathbf{A}$ but an indefinite number of $\vec{b}$'s which might not be known in advance. For example, the $\vec{b}$'s might represent an endless series of measurement outcomes, with $\mathbf{A}$ representing some fixed experimental configuration. We would like to efficiently calculate $\vec{x}$ for each $\vec{b}$ that arrives. If this is done with Gaussian elimination, each calculation would take $O(N^3)$ time.

However, if we can perform an LU decomposition *ahead of time*, then the calculations can be performed much more quickly. The linear equations are

$$\mathbf{L}\,\mathbf{U}\vec{x} = \vec{b}. \tag{5.4.2}$$

This can be broken up into two separate equations:

$$\mathbf{L}\,\vec{y} = \vec{b}, \quad \text{and} \quad \mathbf{U}\vec{x} = \vec{y}. \tag{5.4.3}$$

Because $\mathbf{L}$ is lower-triangular, we can solve the first equation by forward-substitution (similar to back-substitution, except that it goes from the first row to last) to find $\vec{y}$. Then we can solve the second equation by back-substitution, to find $\vec{x}$. The whole process takes $O(N^2)$ time, which is a tremendous improvement over performing a wholesale Gaussian elimination.

However, finding the LU decomposition takes $O(N^3)$ time (we won't go into details here, but it's basically a variant of the row reduction phase of the Gaussian elimination algorithm). Therefore, if we are interested in solving the linear equations only once, or a handful of times, the LU decomposition method does not improve performance. It's useful in situations where the LU decomposition is performed ahead of time. You can think of the LU decomposition as a way of re-arranging the Gaussian elimination algorithm, so that we don't need to know $\vec{b}$ during in the first, expensive $O(N^3)$ phase.

In Python, you can perform the LU decomposition using the `scipy.linalg.lu` function. The forward-substitution and back-substitution steps can be performed using `scipy.linalg.solve_triangular`.

CHAPTER OVERVIEW

6: Eigenvalue Problems

An **eigenvalue problem** is a matrix equation of the form

$$\mathbf{A}\vec{x} = \lambda\vec{x}, \tag{6.1}$$

where $\mathbf{A}$ is a known $N \times N$ matrix. The problem is to find one (or more than one) non-zero vector $\vec{x}$, which is called an **eigenvector**, and the associated $\lambda \in \mathbb{C}$, which is called an **eigenvalue**. Eigenvalue problems are ubiquitous in practically all fields of physics. Most prominently, they are used to describe the "modes" of a physical system, such as the modes of a classical mechanical oscillator, or the energy states of an atom.

Before discussing numerical solutions to the eigenvalue problem, let us quickly review the relevant mathematical facts.

6.1: Basic Facts about Eigenvalue Problems

Even if a matrix $\mathbf{A}$ is real, its eigenvectors and eigenvalues can be complex. For example,

$$\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ i \end{bmatrix} = (1+i) \begin{bmatrix} 1 \\ i \end{bmatrix}. \tag{6.1.1}$$

Eigenvectors are not uniquely defined. Given an eigenvector $\vec{x}$, any nonzero complex multiple of that vector is also an eigenvector of the same matrix, with the same eigenvalue. We can reduce this ambiguity by **normalizing** eigenvectors to a fixed unit length:

$$\sum_{n=0}^{N-1} |x_n|^2 = 1. \tag{6.1.2}$$

Note, however, that even after normalization, there is still an inherent ambiguity in the overall complex phase. Multiplying a normalized eigenvector by any phase factor $e^{i\phi}$ gives another normalized eigenvector with the same eigenvalue.

6.1.1 Matrix Diagonalization

Most matrices are **diagonalizable**, meaning that their eigenvectors span the N-dimensional complex space (where N is the matrix size). Matrices which are not diagonalizable are called **defective**. Many classes of matrices that are relevant to physics (such as Hermitian matrices) are always diagonalizable; i.e., never defective.

The reason for the term "diagonalizable" is as follows. A diagonalizable $N \times N$ matrix $\mathbf{A}$ has eigenvectors that span the N-dimensional space, meaning that we can choose N linearly independent eigenvectors, $\{\vec{x}_0, \vec{x}_1, \cdots \vec{x}_{N-1}\}$, with eigenvalues $\{\lambda_0, \lambda_1, \cdots \lambda_{N-1}\}$. We refer to such a set of N eigenvalues as the "eigenvalues of $\mathbf{A}$". If we group the eigenvectors into an $N \times N$ matrix

$$\mathbf{Q} = [\vec{x}_0, \vec{x}_1, \cdots \vec{x}_{N-1}], \tag{6.1.3}$$

then, since the eigenvectors are linearly independent, $\mathbf{Q}$ is guaranteed to be invertible. Using the eigenvalue equation, we can then show that

$$\mathbf{Q}^{-1}\,\mathbf{A}\,\mathbf{Q} = \begin{bmatrix} \lambda_0 & 0 & \cdots & 0 \\ 0 & \lambda_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_{N-1} \end{bmatrix}. \tag{6.1.4}$$

In other words, there exists a **similarity transformation** which converts $\mathbf{A}$ into a diagonal matrix. The N numbers along the diagonal are precisely the eigenvalues of $\mathbf{A}$.

6.1.2 The Characteristic Polynomial

One of the most important consequences of diagonalizability is that the determinant of a diagonalizable matrix $\mathbf{A}$ is the product of its eigenvalues:

$$\det(\mathrm{A}) = \prod_{n=0}^{N-1} \lambda_n \tag{6.1.5}$$

This can be proven by taking the determinant of the similarity transformation equation, and using (i) the property of the determinant that $\det(\mathbf{UV}) = \det(\mathbf{U})\det(\mathbf{V})$, and (ii) the fact that the determinant of a diagonal matrix is the product of the elements along the diagonal.

In particular, the determinant of $\mathbf{A}$ is zero if one of its eigenvalues is zero. This fact can be further applied to the following re-arrangement of the eigenvalue equation:

$$\left(\mathbf{A} - \lambda \mathbf{I}\right) \vec{x} = 0, \tag{6.1.6}$$

where $\mathbf{I}$ is the $N \times N$ identity matrix. This says that the matrix $\mathbf{A} - \lambda \mathbf{I}$ has an eigenvalue of zero, meaning that for any eigenvalue λ,

$$\det(\mathbf{A} - \lambda \mathbf{I}) = 0. \tag{6.1.7}$$

The left-hand side of the above equation is a polynomial in the variable λ, of degree N. This is called the **characteristic polynomial** of the matrix $\mathbf{A}$. Its roots are eigenvalues of $\mathbf{A}$, and vice versa.

For 2×2 matrices, the standard way of calculating the eigenvalues is to find the roots of the characteristic polynomial. However, this is not a reliable method for finding the eigenvalues of larger matrices. There is a well-known and important result in mathematics, known as Abel's impossibility theorem, which states that polynomials of degree 5 and higher have no general algebraic solution. (By comparison, degree-2 polynomials have a general algebraic solution, which is the familiar quadratic formula, and similar formulas exist for degree-3 and degree-4 polynomials.) A matrix of size $N \geq 5$ has a characteristic polynomial of degree $N \geq 5$, and Abel's impossibility theorem tells us that we can't calculate the roots of that characteristic polynomial by ordinary arithmetic.

In fact, Abel's impossibility theorem leads to an even stronger conclusion: there is no general algebraic method for finding the eigenvalues of a matrix of size $N \geq 5$, whether using the characteristic polynomial *or any other method*. For suppose we had such a method for finding the eigenvalues of a matrix. Then, for any polynomial equation of degree $N \geq 5$, of the form

$$a_0 + a_1\lambda + \cdots + a_{N-1}\lambda^{N-1} + \lambda^N = 0, \tag{6.1.8}$$

we can construct an $N \times N$ "companion matrix" of the form

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \ddots & 1 \\ -a_0 & -a_1 & -a_2 & \cdots & -a_{N-1} \end{bmatrix}. \tag{6.1.9}$$

As you can check for yourself, each root λ of the polynomial is also an eigenvalue of the companion matrix, with corresponding eigenvector

$$\vec{x} = \begin{bmatrix} 1 \\ \lambda \\ \vdots \\ \lambda^{N-1} \end{bmatrix}. \tag{6.1.10}$$

Hence, if there exists a general algebraic method for finding the eigenvalues of a large matrix, that would allow us to find solve polynomial equations of high degree. Abel's impossibility theorem tells us that no such solution method can exist.

This might seem like a terrible problem, but in fact there's a way around it, as we'll shortly see.

6.1.3 Hermitian Matrices

A **Hermitian** matrix $\mathbf{H}$ is a matrix which has the property

$$\mathbf{H}^\dagger = \mathbf{H}, \tag{6.1.11}$$

where $\mathbf{H}^\dagger$ denotes the "Hermitian conjugate", which is matrix transposition accompanied by complex conjugation:

$$\mathbf{H}^\dagger \equiv \left(\mathbf{H}^T\right)^*, \quad \text{i.e. } \left(H^\dagger\right)_{ij} = H^*_{ji}. \tag{6.1.12}$$

Hermitian matrices have the nice property that all their eigenvalues are real. This can be easily proven using index notation:

$$\sum_j H_{ij}x_j = \lambda x_i \quad \Rightarrow \quad \sum_j x_j^* H_{ji} = \lambda^* x_i^* \tag{6.1.13}$$

$$\Rightarrow \sum_{ij} x_i^* H_{ij} x_j = \lambda \sum_i |x_i|^2 = \lambda^* \sum_j |x_j|^2 \tag{6.1.14}$$

$$\Rightarrow \lambda = \lambda^*. \tag{6.1.15}$$

In quantum mechanics, Hermitian matrices play a special role: they represent measurement operators, and their eigenvalues (which are restricted to the real numbers) are the set of possible measurement outcomes.

6.2: Numerical Eigensolvers

As discussed above, Abel's impossibility theory tells us that there is no general algebraic formula for calculating the eigenvalues of an $N \times N$ matrix, for $N \geq 5$. In practice, however, there exist numerical methods, called **eigensolvers**, which can compute eigenvalues (and eigenvectors) even for very large matrices, with hundreds of rows/columns, or larger. How could this be?

The answer is that numerical eigensolvers are *approximate, not exact*. But even though their results are not exact, they are *very* precise—they can approach the exact eigenvalues to within the fundamental precision limits of floating-point arithmetic. Since we're limited by those floating-point precision limits anyway, that's good enough!

6.2.1 Sketch of the Eigensolver Method

We will not go into detail about how numerical eigensolvers work, as that would involve a rather long digression. For those interested, the following paper provides a good pedagogical presentation of the subject:

- Bruno Lang, *Direct Solvers for Symmetric Eigenvalue Problems*. Modern Methods and Algorithms of Quantum Chemistry, Proceedings, Second Ed. (2000). PDF download link

Here is a *very* brief sketch of the basic method. Similar to Gaussian elimination, the algorithm contains two phases, a relatively costly/slow initial phase and a relatively fast second phase. The first phase, which is called Householder reduction, applies a carefully-chosen set of similarity transformations to the input matrix $\mathbf{A}_0$:

$$\mathbf{A}_0 \;\rightarrow\; \mathbf{A}_1 = \mathbf{X}_1^{-1}\mathbf{A}_0\mathbf{X}_1 \;\rightarrow\; \mathbf{A}_2 = \mathbf{X}_2^{-1}\mathbf{A}_1\mathbf{X}_2, \;\; \text{etc.} \tag{6.2.1}$$

The end result is a matrix $\mathbf{A}_k$ which is in Hessenberg form: the elements below the first subdiagonal are all zero (the elements immediately below the main diagonal, i.e. along the first subdiagonal, are allowed to be nonzero). The entire Householder reduction phase requires $O(N^3)$ arithmetic operations, where N is the size of the matrix.

The second phase of the algorithm is called QR iteration. Using a different type of similarity transformation, the elements along the subdiagonal of the Hessenberg matrix are reduced in magnitude. When these elements become negligible, the matrix becomes upper-triangular; in that case, the eigenvalues are simply the elements along the diagonal.

The QR process is *iterative*, in that it progressively reduces the magnitude of the matrix elements along the subdiagonal. Formally, an infinite number of iterations would be required to reduce these elements to zero—that's why Abel's impossibility theorem isn't violated! In practice, however, QR iteration converges extremely quickly, so this phase ends up taking only $O(N^2)$ time.

Hence, the overall runtime for finding the eigenvalues of a matrix scales as $O(N^3)$. The eigenvectors can also be computed as a side-effect of the algorithm, with no extra increase in the runtime scaling.

6.2.2 Python Implementation

There are four main numerical eigensolvers implemented in Scipy, which are all found in the `scipy.linalg` package:

- `scipy.linalg.eig` returns the eigenvalues and eigenvectors of a matrix.
- `scipy.linalg.eigvals` returns the eigenvalues (only) of a matrix.
- `scipy.linalg.eigh` returns the eigenvalues and eigenvectors of a Hermitian matrix.
- `scipy.linalg.eigvalsh` returns the eigenvalues (only) of a Hermitian matrix.

The reason for having four separate functions is efficiency. The runtimes of all four functions *scale* as $O(N^3)$, but for each N the actual runtimes of `eigvals` and `eigvalsh` will be shorter than `eig` and `eigh`, because the eigensolver is only asked to find the eigenvalues and need not construct the eigenvectors. Furthermore, `eigvalsh` is faster than `eigvals`, and `eigh` is faster than `eig`, because the eigensolver algorithm can make use of certain numerical shortcuts which are valid only for Hermitian matrices.

If you pass `eigvalsh` or `eigh` a matrix that is not actually Hermitian, the results are unpredictable; the function may return a wrong value without signaling any error. Therefore, you should only use these functions if you are sure that the input matrix is definitely Hermitian (which is usually because you constructed the matrix that way); if the mtrix is Hermitian, `eigvalsh` or `eigh` are certainly preferable to use, because they run faster than their non-Hermitian counterparts.

Here is a short program that uses `eigvals` to find the eigenvalues of a 3×3 matrix:

```
from scipy import *
import scipy.linalg as lin

A = array([[1,3,1],[1, 3, 4],[2, 4, 2]])
lambd = lin.eigvals(A)

print(lambd)
```

Running the program outputs:

```
[ 7.45031849+0.j          -0.72515925+0.52865751j -0.72515925-0.52865751j]
```

The return value of `eigvals` is a 1D array of complex numbers, storing the eigenvalues of the input. The `eigvalsh` function behaves similarly, except that a real array is returned (since Hermitian matrices have real eigenvalues). Note: we cannot use `lambda` as the name of a variable, because `lambda` is reserved as a special keyword in Python.

Here is an example of using `eig` :

```
>>> A = array([[1,3,1],[1, 3, 4],[2, 4, 2]])
>>> lambd, Q = lin.eig(A)
>>> lambd
array([ 7.45031849+0.j    , -0.72515925+0.52865751j,   -0.72515925-0.52865751j])
>>> Q
array([[ 0.40501343+0.j        ,  0.73795979+0.j        ,  0.73795979-0.j        ],
       [ 0.65985810+0.j        , -0.51208724+0.22130102j, -0.51208724-0.22130102j],
       [ 0.63289132+0.j        ,  0.26316357-0.27377508j,  0.26316357+0.27377508j]])
```

The `eig` function returns a pair of values; the first is a 1D array of eigenvalues (which we name `lambd` in the above example), and the second is a 2D array containing the corresponding eigenvectors in each column (which we name `Q`). For example, the first eigenvector can be accessed with `Q[:,0]` . We can verify that this is indeed an eigenvector:

```
>>> dot(A, Q[:,0])
array([ 3.01747903+0.j,  4.91615298+0.j,  4.71524187+0.j])
>>> lambd[0] * Q[:,0]
array([ 3.01747903+0.j,  4.91615298+0.j,  4.71524187+0.j])
```

The `eigh` function behaves similarly, except that the 1D array of eigenvalues is real.

6.2.3 Generalized Eigenvalue Problem

Sometimes, you might also come across **generalized eigenvalue problems**, which have the form

$$\mathbf{A}\,\vec{x} = \lambda\,\mathbf{B}\,\vec{x}, \tag{6.2.2}$$

for known equal-sized square matrices $\mathbf{A}$ and $\mathbf{B}$. We call λ a "generalized eigenvalue", and $\vec{x}$ a "generalized eigenvector", for the pair of matrices $(\mathbf{A}, \mathbf{B})$. The generalized eigenvalue problem reduces to the ordinary eigenvalue problem when $\mathbf{B}$ is the identity matrix.

The naive way to solve the generalized eigenvalue problem would be to compute the inverse of $\mathbf{B}^{-1}$, and then solve the eigenvalue problem for $\mathbf{B}^{-1}\mathbf{A}$. However, it turns out that the generalized eigenvalue problem can be solved directly with only slight modifications to the usual numerical eigensolver algorithm. In fact, the Scipy eigensolvers described in the previous section will solve the generalized eigenvalue problem if you pass a 2D array $\mathbf{B}$ as the second input (if that second input is omitted, the eigensolvers solve the ordinary eigenvalue problem, as described above).

Here is an example program for solving a generalized eigenvalue problem:

```
from scipy import *
import scipy.linalg as lin

A = array([[1,3,1],[1, 3, 4],[2, 4, 2]])
B = array([[0,2,1], [0, 1, 1], [2, 0, 1]])

lambd, Q = lin.eig(A, B)

## Verify the solution for first generalized eigenvector:
lhs = dot(A,Q[:,0])             # A . x
rhs = lambd[0] * dot(B, Q[:,0]) # lambda B . x

print(lhs)
print(rhs)
```

Running the above program prints:

```
[-0.16078694+0.j -0.07726949+0.j  0.42268561+0.j]
[-0.16078694+0.j -0.07726949+0.j  0.42268561+0.j]
```

The Hermitian eigensolvers, `eigh` and `eigvalsh`, can be used to solve the generalized eigenvalue problem only if *both* the $\mathbf{A}$ and $\mathbf{B}$ matrices are Hermitian.

CHAPTER OVERVIEW

7: Finite-Difference Equations

One of the most common tasks in scientific computing is finding solutions to differential equations, because most physical theories are formulated using differential equations. In classical mechanics, for example, a mechanical system is described by a second-order differential equation in time (Newton's second law); and in classical electromagnetism, the electromagnetic fields are described by first-order partial differential equations in space and time (Maxwell's equations).

In order to describe continuous functions (and the differential equations that act on them), computational schemes usually adopt the strategy of **discretization**. Consider a general mathematical function of one real variable, $\psi(x)$, where the domain of the input is $\mathbb{R}$, or some finite interval. In principle, in order to fully specify the function, we have to enumerate its values for all possible inputs x; but since x can vary continuously, the set is uncountably infinite, so such an enumeration is impossible on a digital computer with finite discrete memory. What we can do, instead, is to enumerate the function's values at a finite and discrete set of points,

$$\{x_n \mid n = 0, 1, 2, \ldots, N-1\}. \tag{7.1}$$

We define the values at these points as

$$\psi_n \equiv \psi(x_n). \tag{7.2}$$

If x_n is appropriately chosen, the set of values $\{\psi_n\}$ ought to describe $\psi(x)$ quite accurately. One reason for this is that physical theories typically involve differential equations of low order (e.g., first, second, or third order, rather than, say, order $1,000,000$). Hence, if the discretization points are sufficiently finely-spaced, the value of the function, *and all its higher-order derivatives*, will vary only slightly between discretization points.

As we shall see, discretization converts differential equations into discrete systems of equations, called **finite-difference equations**. These can then be solved using the standard methods of numerical linear algebra.

7.1: Derivatives

Suppose we have discretized a function of one variable, obtaining a set of $\psi_n \equiv \psi(x_n)$ as described above. For simplicity, we assume that the discretization points are evenly-spaced and arranged in increasing order (this is the simplest and most common discretization scheme). The spacing between points is defined as

$$h \equiv x_{n+1} - x_n. \tag{7.1.1}$$

Let us discuss how the first and higher-order derivatives of $\psi(x)$ can be represented under discretization.

7.1.1 First Derivative

The most straightforward representation of the first derivative is the **forward-difference** formula:

$$\psi'(x_n) \approx \frac{\psi_{n+1} - \psi_n}{h} \tag{7.1.2}$$

This is inspired by the usual definition of the derivative of a function, and approaches the true derivative as $h \to 0$. However, it is not a very good approximation. To see why, let's analyze the **error** in the formula, which is defined as the absolute value of the difference between the formula and the exact value of the derivative:

$$\mathcal{E} = \left| \psi'(x_n) - \frac{\psi_{n+1} - \psi_n}{h} \right| \tag{7.1.3}$$

We can expand ψ_{n+1} in a Taylor series around x_n:

$$\psi_{n+1} = \psi_n + h\,\psi'(x_n) + \frac{h^2}{2}\psi''(x_n) + \frac{h^3}{6}\psi'''(x_n) + O(h^4) \tag{7.1.4}$$

Plugging this into the error formula, we find that the error decreases linearly with the spacing:

$$\mathcal{E} = \left| \frac{h}{2}\psi''(x_n) + O(h^2) \right| \sim O(h). \tag{7.1.5}$$

There is a better alternative, called the **mid-point** formula. This approximates the first derivative by sampling the points to the left *and* right of the desired position:

$$\psi'(x_n) \approx \frac{\psi_{n+1} - \psi_{n-1}}{2h}. \tag{7.1.6}$$

To see why this is better, let us write down the Taylor series for $\psi_{n\pm1}$:

$$\psi_{n+1} = \psi_n + h\,\psi'(x_n) + \frac{h^2}{2}\psi''(x_n) + \frac{h^3}{6}\psi'''(x_n) + \frac{h^4}{24}\psi''''(x_n) + O(h^5) \tag{7.1.7}$$

$$\psi_{n-1} = \psi_n - h\,\psi'(x_n) + \frac{h^2}{2}\psi''(x_n) - \frac{h^3}{6}\psi'''(x_n) + \frac{h^4}{24}\psi''''(x_n) + O(h^5) \tag{7.1.8}$$

Note that the two series have the same terms involving even powers of h, whereas the terms involving odd powers of h have opposite signs. Hence, if we subtract the second series from the first, the result is

$$\psi_{n+1} - \psi_{n-1} = 2h\,\psi'(x_n) + 2\frac{h^3}{6}\psi'''(x_n) + O(h^5) \tag{7.1.9}$$

Because the $O(h^2)$ terms are equal in the two series, they cancel out under subtraction, and only the $O(h^3)$ and higher terms survive. After re-arranging the above equation, we get

$$\psi'(x_n) = \frac{\psi_{n+1} - \psi_{n-1}}{2h} + O(h^2). \tag{7.1.10}$$

Hence, the error of the mid-point formula scales as $O(h^2)$, which is a good improvement over the $O(h)$ error of the forward-difference formula. What's especially nice is that the mid-point formula requires the same number of arithmetic operations to calculate as the forward-difference formula, so this is a free lunch!

It is possible to come up with better approximation formulas for the first derivative by including terms involving $\psi_{n\pm2}$ etc., with the goal of canceling the $O(h^3)$ or higher terms in the Taylor series. For most practical purposes, however, the mid-point rule is sufficient.

7.1.2 Second Derivative

The discretization of the second derivative is easy to figure out too. We again write down the Taylor series for $\psi_{n\pm1}$:

$$\psi_{n+1} = \psi_n + h\,\psi'(x_n) + \frac{h^2}{2}\psi''(x_n) + \frac{h^3}{6}\psi'''(x_n) + \frac{h^4}{24}\psi''''(x_n) + O(h^5) \tag{7.1.11}$$

$$\psi_{n-1} = \psi_n - h\,\psi'(x_n) + \frac{h^2}{2}\psi''(x_n) - \frac{h^3}{6}\psi'''(x_n) + \frac{h^4}{24}\psi''''(x_n) + O(h^5) \tag{7.1.12}$$

When we add the two series together, the terms involving odd powers of h cancel, and the result is

$$\psi_{n+1} + \psi_{n-1} = 2\psi_n + h^2\psi''(x_n) + \frac{h^4}{12}\psi''''(x_n) + O(h^5). \tag{7.1.13}$$

A minor rearrangement of the equation then gives

$$\psi''(x_n) \approx \frac{\psi_{n+1} - 2\psi_n + \psi_{n-1}}{h^2} + O(h^2). \tag{7.1.14}$$

This is called the **three-point rule** for the second derivative, because it involves the value of the function at the three points x_{n+1}, x_n, and x_{n-1}.

7.2: Discretizing Partial Differential Equations

With discretized derivatives, differential equations can be formulated as discrete systems of equations. We will discuss this using a specific example: the discretization of the time-independent Schrödinger wave equation in 1D.

7.2.1 Deriving a Finite-Difference Equation

The 1D time-independent Schrödinger wave equation is the second-order ordinary differential equation

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2}+V(x)\psi(x)=E\psi(x), \tag{7.2.1}$$

where $\hbar$ is Planck's constant divided by 2π, m is the mass of the particle, $V(x)$ is the potential, $\psi(x)$ is the quantum wavefunction of an energy eigenstate of the particle, and E is the corresponding energy. The differential equation is usually treated as an eigenproblem, in the sense that we are given $V(x)$ and seek to find the possible values of the eigenfunction $\psi(x)$ and the energy eigenvalue E. For convenience, we will adopt units where $\hbar=m=1$:

$$-\frac{1}{2}\frac{d^2\psi}{dx^2}+V(x)\psi(x)=E\psi(x). \tag{7.2.2}$$

To discretize this differential equation, we simply evaluate it at $x=x_n$:

$$-\frac{1}{2}\psi''(x_n)+V_n\psi_n=E\psi_n, \tag{7.2.3}$$

where, for conciseness, we denote

$$V_n\equiv V(x_n). \tag{7.2.4}$$

We then replace the second derivative $\psi''(x_n)$ with a discrete approximation, specifically the three-point rule:

$$-\frac{1}{2h^2}\left[\psi_{n+1}-2\psi_n+\psi_{n-1}\right]+V_n\psi_n=E\psi_n. \tag{7.2.5}$$

This result is called a **finite-difference equation**, and it would be valid for all n if the number of discretization points is infinite. However, if there is a finite number of discretization points, $\{x_0,x_1,\ldots,x_{N-1}\}$, then the finite-difference formula fails at the boundary points, $n=0$ and $n=N-1$, where it involves the value of the function at the "non-existent" points x_{-1} and x_N. We'll see how to handle this problem in the next section.

Boundaries aside, the finite-difference equation describes a matrix equation:

$$\left\{-\frac{1}{2h^2}\begin{bmatrix}\ddots & \ddots & & & & \\ \ddots & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & \ddots & \\ & & & \ddots & \ddots\end{bmatrix}+\begin{bmatrix}\ddots & & & & & \\ & V_{n-1} & & & & \\ & & V_n & & & \\ & & & V_{n+1} & & \\ & & & & \ddots\end{bmatrix}\right\}\begin{bmatrix}\vdots\\ \psi_{n-1}\\ \psi_n\\ \psi_{n+1}\\ \vdots\end{bmatrix}=E\begin{bmatrix}\vdots\\ \psi_{n-1}\\ \psi_n\\ \psi_{n+1}\\ \vdots\end{bmatrix}. \tag{7.2.6}$$

The second-derivative operator is represented by a tridiagonal matrix with -2 in each diagonal element, and 1 in the elements directly above and below the diagonal. The potential operator is represented by a diagonal matrix, where the elements along the diagonal are the values of the potential at each discretization point. In this way, the Schrödinger wave equation is reduced to a discrete eigenvalue problem.

7.2.2 Boundary Conditions

We now have to figure out how to handle the boundaries. Let us suppose $\psi(x)$ is defined over a finite interval, $a\le x\le b$. As we recall from the theory of differential equations, the solution to a differential equation is not wholly determined by the differential equation itself, but also by the boundary conditions that are imposed. Thus, we have to specify how $\psi(x)$ behaves at the end-points of the interval. We will show how this is done for a couple of the most common boundary conditions; other choices of boundary conditions can be handled using the same kind of reasoning.

Dirichlet Boundary Conditions

Under **Dirichlet boundary conditions**, the wavefunction vanishes at the boundaries:

$$\psi(a) = \psi(b) = 0. \tag{7.2.7}$$

Physically, these boundary conditions apply if we let the potential blow up in the external regions, $x > b$ and $x < a$, thus forcing the wavefunction to be strictly confined to the interval $a \leq x \leq b$.

We have not yet stated how the discretization points $\{x_0, \ldots, x_{N-1}\}$ are distributed within the interval; we will make this decision in tandem with the implementation of the boundary conditions. Consider the first discretization point, x_0, wherever it is. The finite-difference equation at this point is

$$-\frac{1}{2h^2}\Big[\psi_{-1} - 2\psi_0 + \psi_1\Big] + V_0\psi_0 = E\psi_0. \tag{7.2.8}$$

This involves the wavefunction at x_{-1}, which lies just outside our set of discretization points. But if we choose the discretization points so that $x_{-1} = a$, then $\psi_{-1} = 0$ under Dirichlet boundary conditions, so the above finite-difference formula reduces to

$$-\frac{1}{2h^2}\Big[-2\psi_0 + \psi_1\Big] + V_0\psi_0 = E\psi_0. \tag{7.2.9}$$

As for the other boundary, the finite-difference equation at x_{N-1} involves ψ_N. If we choose the discretization points so that $x_N = b$, then the finite-difference formula becomes

$$-\frac{1}{2h^2}\Big[\psi_{N-2} - 2\psi_{N-1}\Big] + V_{N-1}\psi_{N-1} = E\psi_{N-1}. \tag{7.2.10}$$

From this, we conclude that the discretization points ought to be equally spaced, with x_0 at a distance h to the right of the left boundary a and x_{N-1} a distance h to the left of the right boundary b. This is shown in the following figure:

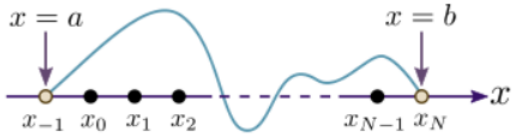

Figure 7.2.1: Position of discretization points for Dirichlet boundary conditions at $x = a$ and $x = b$.

Since there are N discretization points, the interval should contain $(N+1)$ multiples of h. Hence,

$$h = \frac{b-a}{N+1} \quad \Rightarrow \quad x_n = a + h(n+1) = \frac{a(N-n) + b(n+1)}{N+1}. \tag{7.2.11}$$

Having made the above choices, the matrix equation becomes

$$\left\{-\frac{1}{2h^2}\begin{bmatrix} -2 & 1 & & \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{bmatrix} + \begin{bmatrix} V_0 & & & \\ & V_1 & & \\ & & \ddots & \\ & & & V_{N-1} \end{bmatrix}\right\} \begin{bmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{N-1} \end{bmatrix} = E \begin{bmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{N-1} \end{bmatrix}. \tag{7.2.12}$$

You can check for yourself that the first and last rows of this equation are the correct finite-difference equations at the boundary points, corresponding to Dirichlet boundary conditions.

Neuman boundary conditions

Neumann boundary conditions are another common choice of boundary conditions. They state that the first derivatives vanish at the boundaries:

$$\psi'(a) = \psi'(b) = 0. \tag{7.2.13}$$

An example of such a boundary condition is encountered in electrostatics, where the first derivative of the electric potential goes to zero at the surface of a charged metallic surface.

We follow the same strategy as before, figuring out the discretization points in tandem with the boundary conditions. Consider again the finite-difference equation at the first discretization point:

$$-\frac{1}{2h^2}\Big[\psi_{-1}-2\psi_0+\psi_1\Big]+V_0\psi_0=E\psi_0. \tag{7.2.14}$$

To implement the condition that first derivative vanishes at the boundary, we invoke the mid-point rule. Suppose the boundary point $x=a$ falls *in between* the points x_{-1} and x_0. Then, according to the mid-point rule,

$$\frac{\psi_0-\psi_{-1}}{h}\approx\psi'(a)=0. \tag{7.2.15}$$

With this choice, therefore, we can make the replacement $\psi_{-1}=\psi_0$ in the finite-difference equation, which then becomes

$$-\frac{1}{2h^2}\Big[-\psi_0+\psi_1\Big]+V_0\psi_0=E\psi_0. \tag{7.2.16}$$

Similarly, to apply the Neumann boundary condition at $x=b$, we let the boundary fall between x_{N-1} and x_N, so that the finite-difference equation becomes

$$-\frac{1}{2h^2}\Big[\psi_{N-2}-\psi_{N-1}\Big]+V_{N-1}\psi_{N-1}=E\psi_{N-1}. \tag{7.2.17}$$

The resulting distribution of discretization points is shown in the following figure:

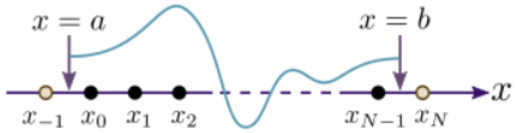

Figure 7.2.2: Position of discretization points for Neumann boundary conditions at $x=a$ and $x=b$.

Unlike the Dirichlet case, the interval contains N multiples of h. Hence, we get a different formula for the positions of the discretization points

$$h=\frac{b-a}{N}\quad\Rightarrow\quad x_n=a+h\left(n+\frac{1}{2}\right)=\frac{a(N-n-\frac{1}{2})+b(n+\frac{1}{2})}{N}. \tag{7.2.18}$$

The matrix equation is:

$$\left\{-\frac{1}{2h^2}\begin{bmatrix}-1&1&&&\\1&-2&\ddots&&\\&\ddots&\ddots&\ddots&\\&&\ddots&-2&1\\&&&1&-1\end{bmatrix}+\begin{bmatrix}V_0&&&\\&V_1&&\\&&\ddots&\\&&&V_{N-1}\end{bmatrix}\right\}\begin{bmatrix}\psi_0\\\psi_1\\\vdots\\\psi_{N-1}\end{bmatrix}=E\begin{bmatrix}\psi_0\\\psi_1\\\vdots\\\psi_{N-1}\end{bmatrix}. \tag{7.2.19}$$

Due to the Neumann boundary conditions and the mid-point rule, the tridiagonal matrix has -1 instead of -2 on its corner entries. Again, you can verify that the first and last rows of this matrix equation correspond to the correct finite-difference equations for the boundary points.

7.3: Higher Dimensions

We can work out the finite-difference equations for higher dimensions in a similar manner. In two dimensions, for example, the wavefunction $\psi(x,y)$ is described with two indices:

$$\psi_{mn} \equiv \psi(x_m, y_n). \tag{7.3.1}$$

The discretization of the derivatives is carried out in the same way, using the mid-point rule for first partial derivatives in each direction, and the three-point rule for the second partial derivative in each direction. Let us suppose that the discretization spacing is equal in both directions:

$$h = x_{m+1} - x_m = y_{n+1} - y_n. \tag{7.3.2}$$

Then, for the second derivative, the Laplacian operator

$$\nabla^2\psi(x,y) \equiv \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} \tag{7.3.3}$$

can be approximated by a **five-point rule**, which involves the value of the function at (m,n) and its four nearest neighbors:

$$\nabla^2\psi(x_m,y_n) \approx \frac{\psi_{m+1,n} + \psi_{m,n+1} - 4\psi_{mn} + \psi_{m-1,n} + \psi_{m,n-1}}{h^2} + O(h^2). \tag{7.3.4}$$

For instance, the finite-difference equations for the 2D Schrödinger wave equation is

$$-\frac{1}{2h^2}\left[\psi_{m+1,n} + \psi_{m,n+1} - 4\psi_{mn} + \psi_{m-1,n} + \psi_{m,n-1}\right] + V_{mn}\psi_{mn} = E\psi_{mn}. \tag{7.3.5}$$

7.3.1 Matrix Reshaping

Higher-dimensional differential equations introduce one annoying complication: in order to convert between the finite-difference equation and the matrix equation, the indices have to be re-organized. For instance, the matrix form of the 2D Schrödinger wave equation should have the form

$$\sum_\nu H_{\mu\nu}\psi_\nu = E\psi_\mu, \tag{7.3.6}$$

where the wavefunctions are organized into a 1D array labeled by a "point index" μ. Each point index corresponds to a *pair* of "grid indices", (m,n), representing spatial coordinates on a 2D grid. We have to be careful not to mix up the two types of indices.

We will adopt the following conversion scheme between point indices and grid indices:

$$\mu(m,n) = mN + n, \quad \text{where } m \in \{0,\ldots,M-1\},\ \ n \in \{0,\ldots,N-1\}. \tag{7.3.7}$$

One good thing about this conversion scheme is that Scipy provides a `reshape` function which can convert a 2D array with grid indices (m,n) into a 1D array with the point index μ:

```
>>> a = array([[0,1,2],[3,4,5],[6,7,8]])
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b = reshape(a, (9))     # Reshape a into a 1D array of size 9
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

The `reshape` function can also convert a 1D back into the 2D array, in the right order:

```
>>> c = reshape(b, (3,3))   # Reshape b into a 2D array of size 3x3
>>> c
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Under point indices, the discretized derivatives take the following forms:

$$\frac{\partial \psi}{\partial x}(\vec{r}_\mu) \approx \frac{1}{2h}(\psi_{\mu+N} - \psi_{\mu-N}) \tag{7.3.8}$$

$$\frac{\partial \psi}{\partial y}(\vec{r}_\mu) \approx \frac{1}{2h}(\psi_{\mu+1} - \psi_{\mu-1}) \tag{7.3.9}$$

$$\nabla^2 \psi(\vec{r}_\mu) \approx \frac{1}{h^2}(\psi_{\mu+N} + \psi_{\mu+1} - 4\psi_\mu + \psi_{\mu-N} + \psi_{\mu-1})\,. \tag{7.3.10}$$

The role of boundary conditions is left as an exercise. There are now two sets of boundaries, at $m \in \{0, M-1\}$ and $n \in \{0, N-1\}$. By examining the finite-difference equations along each boundary, we can (i) assign the right discretization coordinates and (ii) modify the finite-difference matrix elements to fit the boundary conditions. The details are slightly tedious to work out, but the logic is essentially the same as in the previously-discussed 1D cases.

CHAPTER OVERVIEW

8: Sparse Matrices

A **sparse matrix** is a matrix in which most of the entries are zero. Such matrices are very commonly encountered in finite-difference equations. For example, when we discretized the 1D Schrödinger wave equation with Dirichlet boundary conditions, we saw that the Hamiltonian matrix had the tridiagonal form

$$\mathbf{H} = -\frac{1}{2h^2}\begin{bmatrix} -2 & 1 & & \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{bmatrix} + \begin{bmatrix} V_0 & & & \\ & V_1 & & \\ & & \ddots & \\ & & & V_{N-1} \end{bmatrix}. \tag{8.1}$$

Hence, if there are N diagonalization points, the Hamiltonian matrix has a total of N^2 entries, but only $O(N)$ of these entries are non-zero.

8.1: Sparse Matrix Algebra

When matrices are sparse, ordinary approaches to matrix arithmetic are wasteful, since a lot of time is spent adding/subtracting or multiplying by zero. For instance, consider the tridiagonal matrix discussed above. To perform the matrix-vector product $\mathbf{H}\vec{\psi}$ in the usual way, for each row i we must compute

$$\left(\mathbf{H}\vec{\psi}\right)_i = \sum_{j=0}^{N-1} H_{ij}\psi_j \tag{8.1.1}$$

$$= \cdots + (0 \cdot \psi_{i-2}) + (H_{i,i-1} \cdot \psi_{i-1}) + (H_{ii} \cdot \psi_i) + (H_{i,i+1} \cdot \psi_{i+1}) + (0 \cdot \psi_{i+2}) + \cdots \tag{8.1.2}$$

The sum involves $O(N)$ arithmetic operations, so the overall runtime for all N rows is $O(N^2)$. Clearly, however, most of this time is spent multiplying zero elements of $\mathbf{H}$ with elements of $\vec{\psi}$, which doesn't contribute to the final result. If we could omit these terms from each sum, the overall runtime for the product would be only $O(N)$.

However, such savings cannot be achieved with the array data structures we have been using so far. To calculate the matrix-vector product efficiently, the processor needs to know which elements on each row of $\mathbf{H}$ are non-zero, so that it can skip the rest. However, arrays do not provide a fast way to identify which elements are non-zero; the only way to find out is to search the storage blocks one by one, which takes $O(N)$ time on each row. That would wipe out the runtime savings.

Scipy provides special data structures for storing sparse matrices. Unlike ordinary arrays, these data structures are designed so that zero elements are omitted, which not only saves memory, but also allows certain matrix operations to be greatly sped up. Unlike arrays, which can represent not just matrices (2D arrays) but also vectors (1D arrays) and higher-rank tensors (arrays with dimension higher than 2), these sparse data structures are specifically restricted to matrices.

But here's an important catch: there is no single sparse matrix data structure that is ideal for every scenario. Instead, there are multiple **sparse matrix formats**, and each format is only effective for a certain subset of matrix operations, or for certain kinds of sparse matrices. Therefore, we need to know how the different sparse matrix formats are implemented, as well as their benefits and limitations. Of the many sparse matrix formats offered by Scipy, we will discuss four: List of Lists (LIL), Diagonal Storage (DIA), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC).

8.2: Sparse Matrix Formats

8.2.1 List of Lists (LIL)

The **List of Lists** sparse matrix format (which is, for some unknown reason, abbreviated as LIL rather than LOL) is one of the simplest sparse matrix formats. It is shown schematically in the figure below. Each non-zero matrix row is represented by an element in a kind of list structure called a "linked list". Each element in the linked list records the row number, and the column data for the matrix entries in that row. The column data consists of a list, where each list element corresponds to a non-zero matrix element, and stores information about (i) the column number, and (ii) the value of the matrix element.

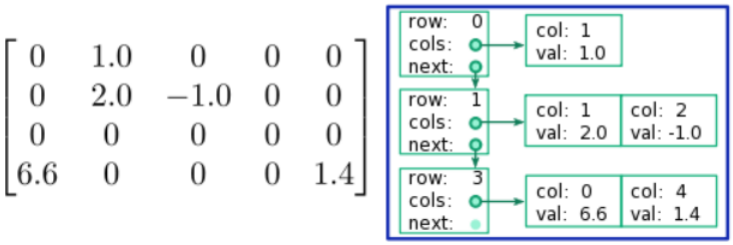

Figure 8.2.1: A sparse matrix and its representation in List-of-Lists (LIL) format.

Evidently, this format is pretty memory-efficient. The list of rows only needs to be as long as the number of non-zero matrix rows; the rest are omitted. Likewise, each list of column data only needs to be as long as the number of non-zero elements on that row. The total amount of memory required is proportional to the number of non-zero elements, regardless of the size of the matrix itself.

Compared to the other sparse matrix formats which we'll discuss, accessing an individual matrix element in LIL format is relatively slow. This is because looking up a given matrix index (i, j) requires stepping through the row list to find an element with row index i; and if one is found, stepping through the column row to find index j. Thus, for example, looking up an element in a diagonal $N \times N$ matrix in the LIL format takes $O(N)$ time! As we'll see, the CSR and CSC formats are much more efficient at element access. For the same reason, matrix arithmetic in the LIL format is very inefficient.

One advantage of the LIL format, however, is that it is relatively easy to alter the "sparsity structure" of the matrix. To add a new non-zero element, one simply has to step through the row list, and either (i) insert a new element into the linked list if the row was not previously on the list (this insertion takes $O(1)$ time), or (ii) modify the column list (which is usually very short if the matrix is very sparse).

For this reason, the LIL format is preferred if you need to construct a sparse matrix where the non-zero elements are not distributed in any useful pattern. One way is to create an empty matrix, then fill in the elements one by one, as shown in the following example. The LIL matrix is created by the `lil_matrix` function, which is provided by the `scipy.sparse` module.

Here is an example program which constructs a LIL matrix, and prints it:

```
from scipy import *
import scipy.sparse as sp

A = sp.lil_matrix((4,5))     # Create empty 4x5 LIL matrix
A[0,1] = 1.0
A[1,1] = 2.0
A[1,2] = -1.0
A[3,0] = 6.6
A[3,4] = 1.4

## Verify the matrix contents by printing it
print(A)
```

When we run the above program, it displays the non-zero elements of the sparse matrix:

```
 (0, 1) 1.0
 (1, 1) 2.0
 (1, 2) -1.0
 (3, 0) 6.6
 (3, 4) 1.4
```

You can also convert the sparse matrix into a conventional 2D array, using the `toarray` method. Suppose we replace the line `print(A)` in the above program with

```
B = A.toarray()
print(B)
```

The result is:

```
[[ 0.   1.   0.   0.   0. ]
 [ 0.   2.  -1.   0.   0. ]
 [ 0.   0.   0.   0.   0. ]
 [ 6.6  0.   0.   0.   1.4]]
```

Note: be careful when calling `toarray` in actual code. If the matrix is huge, the 2D array will eat up unreasonable amounts of memory. It is not uncommon to work with sparse matrices with sizes on the order of $10^5 \times 10^5$, which can take up less an 1MB of memory in a sparse format, but around 80 GB of memory in array format!

8.2.2 Diagonal Storage (DIA)

The Diagonal Storage (DIA) format stores the contents of a sparse matrix along its diagonals. It makes use of a 2D array, which we denote by `data`, and a 1D integer array, which we denote by `offsets`. A typical example is shown in the following figure:

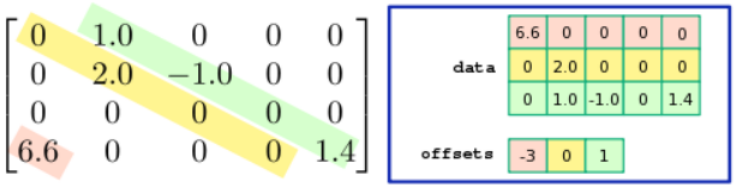

Figure 8.2.2: A sparse matrix and its representation in Diagonal Storage (DIA) format.

Each row of the `data` array stores one of the diagonals of the matrix, and `offsets[i]` records which diagonal that row of the `data` corresponds to, with "offset 0" corresponding to the main diagonal. For instance, in the above example, row 0 of `data` contains the entries `[6.6,0,0,0,0]`), and `offsets[0]` contains the value -3, indicating that the entry 6.6 occurs along the -3 subdiagonal, in column 0. (The extra elements in that row of `data` lie outside the bounds of the matrix, and are ignored.) Diagonals containing only zero are omitted.

For sparse matrices with very few non-zero diagonals, such as diagonal or tridiagonal matrices, the DIA format allows for very quick arithmetic operations. Its main limitation is that looking up each matrix element requires performing a blind search through the `offsets` array. That's fine if there are very few non-zero diagonals, as `offsets` will be small. But if the number of non-zero diagonals becomes large, performance becomes very poor. In the worst-case scenario of an anti-diagonal matrix, element lookup takes $O(N)$ time!

You can create a sparse matrix in the DIA format, using the `dia_matrix` function, which is provided by the `scipy.sparse` module. Here is an example program:

```
from scipy import *
import scipy.sparse as sp
```

```
N = 6  # Matrix size

diag0 = -2 * ones(N)
diag1 = ones(N)

A = sp.dia_matrix(([diag1, diag0, diag1], [-1,0,1]), shape=(N,N))

## Verify the matrix contents by printing it
print(A.toarray())
```

Here, the first input to `dia_matrix` is a tuple of the form `(data, offsets)`, where `data` and `offsets` are arrays of the sort described above. This returns a sparse matrix in the DIA format, with the specified contents (the elements in `data` which lie outside the bounds of the matrix are ignored). In this example, the matrix is tridiagonal with -2 along the main diagonal and 1 along the +1 and -1 diagonals. Running the above program prints the following:

```
[[-2.  1.  0.  0.  0.  0.]
 [ 1. -2.  1.  0.  0.  0.]
 [ 0.  1. -2.  1.  0.  0.]
 [ 0.  0.  1. -2.  1.  0.]
 [ 0.  0.  0.  1. -2.  1.]
 [ 0.  0.  0.  0.  1. -2.]]
```

Another way to create a DIA matrix is to first create a matrix in another format (e.g. a conventional 2D array), and provide that as the input to `dia_matrix`. This returns a sparse matrix with the same contents, in DIA format.

8.2.3 Compressed Sparse Row (CSR)

The **Compressed Sparse Row** (CSR) format represents a sparse matrix using three arrays, which we denote by `data`, `indices`, and `indptr`. An example is shown in the following figure:

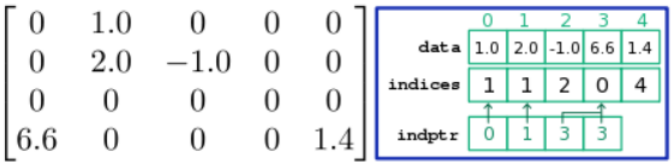

Figure 8.2.3: A sparse matrix and its representation in Compressed Sparse Row (CSR) format.

The array denoted `data` stores the values of the non-zero elements of the matrix, in sequential order from left to right along each row, then from the top row to the bottom. The array denoted `indices` records the *column* index for each of these elements. In the above example, `data[3]` stores a value of 6.6, and `indices[3]` has a value of 0, indicating that a matrix element with value 6.6 occurs in column 0. These two arrays have the same length, equal to the number of non-zero elements in the sparse matrix.

The array denoted `indptr` (which stands for "index pointer") provides an association between the row indices and the matrix elements, but in an indirect manner. Its length is equal to the number of matrix rows (including zero rows). For each row i, if the row is non-zero, `indptr[i]` records the index in the `data` and `indices` arrays corresponding to the first non-zero element on row i. (For a zero row, `indptr` records the index of the next non-zero element occurring in the matrix.)

For example, consider looking up index $(1, 2)$ in the above figure. The row index is 1, so we examine `indptr[1]` (whose value is 1) and `indptr[2]` (whose value is 3). This means that the non-zero elements for matrix row 1 correspond to indices $1 \leq n < 3$ of the `data` and `indices` arrays. We search `indices[1]` and `indices[2]`, looking for a column index of 2. This is found in `indices[2]`, so we look in `data[2]` for the value of the matrix element, which is -1.0.

It is clear that looking up an individual matrix element is very efficient. Unlike the LIL format, where we need to step through a linked list, in the CSR format the `indptr` array lets us to jump straight to the data for the relevant row. For the same reason, the

CSR format is efficient for row slicing operations (e.g., `A[4,:]`), and for matrix-vector products like $\mathbf{A}\vec{x}$ (which involves taking the product of each matrix row with the vector $\vec{x}$).

The CSR format does have several downsides. Column slicing (e.g. `A[:,4]`) is inefficient, since it requires searching through all elements of the `indices` array for the relevant column index. Changes to the sparsity structure (e.g., inserting new elements) are also very inefficient, since all three arrays need to be re-arranged.

To create a sparse matrix in the CSR format, we use the `csr_matrix` function, which is provided by the `scipy.sparse` module. Here is an example program:

```
from scipy import *
import scipy.sparse as sp

data = [1.0, 2.0, -1.0, 6.6, 1.4]
rows = [0, 1, 1, 3, 3]
cols = [1, 1, 2, 0, 4]

A = sp.csr_matrix((data, [rows, cols]), shape=(4,5))
print(A)
```

Here, the first input to `csr_matrix` is a tuple of the form `(data, idx)`, where `data` is a 1D array specifying the non-zero matrix elements, `idx[0,:]` specifies the row indices, and `idx[1,:]` specifies the column indices. Running the program produces the expected results:

```
  (0, 1) 1.0
  (1, 1) 2.0
  (1, 2) -1.0
  (3, 0) 6.6
  (3, 4) 1.4
```

The `csr_matrix` function figures out and generates the three CSR arrays automatically; you don't need to work them out yourself. But if you like, you can inspect the contents of the CSR arrays directly:

```
>>> A.data
array([ 1. ,  2. , -1. ,  6.6,  1.4])
>>> A.indices
array([1, 1, 2, 0, 4], dtype=int32)
>>> A.indptr
array([0, 1, 3, 3, 5], dtype=int32)
```

(There is an extra trailing element of 5 in the `indptr` array. For simplicity, we didn't mention this in the above discussion, but you should be able to figure out why it's there.)

Another way to create a CSR matrix is to first create a matrix in another format (e.g. a conventional 2D array, or a sparse matrix in LIL format), and provide that as the input to `csr_matrix`. This returns the specified matrix in CSR format. For example:

```
>>> A = sp.lil_matrix((6,6))
>>> A[0,1] = 4.0
>>> A[2,0] = 5.0
>>> B = sp.csr_matrix(A)
```

8.2.4 Compressed Sparse Column (CSC)

The **Compressed Sparse Column** (CSC) format is very similar to the CSR format, except that the role of rows and columns is swapped. The `data` array stores non-zero matrix elements in sequential order from top to bottom along each column, then from the left-most column to the right-most. The `indices` array stores *row* indices, and each element of the `indptr` array corresponds to one *column* of the matrix. An example is shown below:

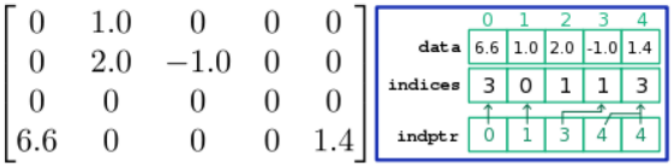

Figure 8.2.4: A sparse matrix and its representation in Compressed Sparse Column (CSC) format.

The CSC format is efficient at matrix lookup, *column* slicing operations (e.g., `A[:,4]`), and vector-matrix products like $\vec{x}^T\mathbf{A}$ (which involves taking the product of the vector $\vec{x}$ with each matrix column). However, it is inefficient for row slicing (e.g. `A[4,:]`), and for changes to the sparsity structure.

To create a sparse matrix in the CSC format, we use the `csc_matrix` function. This is analogous to the `csr_matrix` function for CSR matrices. For example,

```
>>> from scipy import *
>>> import scipy.sparse as sp
>>> data = [1.0, 2.0, -1.0, 6.6, 1.4]
>>> rows = [0, 1, 1, 3, 3]
>>> cols = [1, 1, 2, 0, 4]
>>>
>>> A = sp.csc_matrix((data, [rows, cols]), shape=(4,5))
>>> A
<4x5 sparse matrix of type '<class 'numpy.float64'>'
  with 5 stored elements in Compressed Sparse Column format>
```

8.3: Using Sparse Matrices

Sparse matrix formats should be used, instead of conventional 2D arrays, when dealing sparse matrices of size $10^3 \times 10^3$ or larger. (For matrices of size 100×100 or less, the differences in performance and memory usage are usually negligible.)

Usually, it is good to choose either the CSR or CSC format, depending on what mathematical operations you intend to perform. You can construct the matrix either (i) directly, by means of a `(data, idx)` tuple as described above, or (ii) by creating an LIL matrix, filling in the desired elements, and then converting it to CSR or CSC format.

If you are dealing with a sparse matrix that is "strongly diagonal" (i.e., the non-zero elements occupy a very small number of diagonals, such as a tridiagonal matrix), then you can consider using the DIA format. The main advantage of the DIA format is that it is very easy to construct, by supplying a `(data, offsets)` input to the `dia_matrix` function, as described above. However, the format usually does not perform significantly better than CSR/CSC; and if the matrix is not strongly diagonal, its performance is much worse.

Another common way to construct a sparse matrix is to use the `scipy.sparse.diags` or `scipy.sparse.spdiags` functions. These functions let you specify the contents of the matrix in terms of its diagonals, as well as which sparse format to use. The two functions have slightly different calling conventions; see the documentation for details.

8.3.1 The `dot` method

Each sparse matrix has a `dot` method, which calculates the product of the matrix with an input (in the form of a 1D or 2D array, or sparse matrix), and returns the result. For sparse matrix products, this method should be used instead of the stand-alone function, similarly named `dot`, which is used for non-sparse matrix products. The sparse matrix method makes use of matrix sparsity to speed up the calculation of the product. Typically, the CSR format is faster at this than the other sparse formats.

For example, suppose we want to calculate the product $\mathbf{A}\vec{x}$, where

$$\mathbf{A} = \begin{bmatrix} 0 & 1.0 & 0 & 0 & 0 \\ 0 & 2.0 & -1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 6.6 & 0 & 0 & 0 & 1.4 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 5 \end{bmatrix}. \tag{8.3.1}$$

This could be accomplished with the following program:

```
from scipy import *
import scipy.sparse as sp

data = [1.0, 2.0, -1.0, 6.6, 1.4]
rows = [0, 1, 1, 3, 3]
cols = [1, 1, 2, 0, 4]
A = sp.csr_matrix((data, [rows, cols]), shape=(4,5))
x = array([1.,1.,2.,3.,5.])

y = A.dot(x)

print(y)
```

Running this program gives the expected result:

```
[  1.    0.    0.   13.6]
```

8.3.2 spsolve

The spsolve function, provided in the scipy.sparse.linalg module, is a solver for a sparse linear system of equations. It takes two inputs, $\mathbf{A}$ and $\mathbf{b}$, where $\mathbf{A}$ should be a sparse matrix; $\mathbf{b}$ can be either sparse, or an ordinary 1D or 2D array. It returns the $\mathbf{x}$ which solves the linear equations $\mathbf{x}$. The return value can be either a conventional array or a sparse matrix, depending on whether $\mathbf{b}$ is sparse.

For sparse problems, you should always use spsolve instead of scipy.linalg.solve (the usual solver for non-sparse problems). Here is an example program showing spsolve in action:

```
from scipy import *
import scipy.sparse as sp
import scipy.sparse.linalg as spl

## Make a sparse matrix A and a vector x
data = [1.0, 1.0, 1.0, 1.0, 1.0]
rows = [0, 1, 1, 3, 2]
cols = [1, 1, 2, 0, 3]
A = sp.csr_matrix((data, [rows, cols]), shape=(4,4))
b = array([1.0, 5.0, 3.0, 4.0])

## Solve Ax = b
x = spl.spsolve(A, b)
print(" x = ", x)

## Verify the solution:
print("Ax = ", A.dot(x))
print(" b = ", b)
```

Running the program gives:

```
 x =  [ 4.  1.  4.  3.]
Ax =  [ 1.  5.  3.  4.]
 b =  [ 1.  5.  3.  4.]
```

8.3.3 eigs

For eigenvalue problems involving sparse matrices, one typically does not attempt to find *all* the eigenvalues (and eigenvectors). Sparse matrices are often so huge that solving the full eigenvalue problem would take an impractically long time, even if we receive a speedup from sparse matrix arithmetic. Luckily, in most situations we only need to find a subset of the eigenvalues (and eigenvectors). For example, after discretizing the 1D Schrödinger wave equation, we are normally only interested in the several lowest energy eigenvalues.

The eigs function, provided in the scipy.sparse.linalg module, is an eigensolver for sparse matrices. Unlike the eigensolvers we have previously discussed, such as scipy.linalg.eig , the eigs function only returns a specified subset of the eigenvalues and eigenvectors.

The eigsh function is similar to eigs , except that it is specialized for Hermitian matrices. Both functions make use of a low-level numerical eigensolver library named ARPACK, which is also used by GNU Octave, Matlab, and many other numerical tools. We will not discuss how the algorithm works.

The first input to eigs or eigsh is the matrix for which we want to find the eigenvalues. Several other optional inputs are also accepted. Here are the most commonly-used ones:

- The optional parameter named `k` specifies the number of eigenvalues to find (the default is 6).
- The optional parameter named `M`, if supplied, specifies a right-hand matrix for a generalized eigenvalue problem.
- The optional parameter named `sigma`, if supplied, should be a number; it means to find the `k` eigenvalues which are closest to that number.
- The optional parameter named `which` specifies which eigenvalues to find, using a criteria different from `sigma`: `'LM'` means to find the eigenvalues with the largest magnitudes, `'SM'` means to find those with the smallest magnitudes, etc. You cannot simultaneously specify both `sigma` and `which`. When finding small eigenvalues, it is usually better to use `sigma` instead of `which` (see the discussion in the next section).
- The optional parameter named `return_eigenvectors`, if `True` (the default), means to return both eigenvalues and eigenvectors. If `False`, the function returns the eigenvalues only.

For the full list of inputs, see the full function documentation for `eigs` and `eigsh`.

By default, `eigs` and `eigsh` return two values: a 1D array (which is complex for `eigs` and real for `eigsh`) containing the found eigenvalues, and a 2D array where each column is one of the corresponding eigenvectors. If the optional parameter named `return_eigenvectors` is set to `False`, then only the 1D array of eigenvalues is returned.

In the next section, we will see an example of using `eigsh`.

8.4: Example- Particle-in-a-Box Problem

To demonstrate the use of sparse matrices for solving finite-difference equations, we consider the 1D particle-in-a-box problem. This consists of the 1D time-independent Schrödinger wave equation,

$$-\frac{1}{2}\frac{d^2\psi}{dx^2} = E\psi(x), \quad 0 \le x \le L, \tag{8.4.1}$$

together with the Dirichlet boundary conditions

$$\psi(0) = \psi(L) = 0. \tag{8.4.2}$$

The analytic solution is well known to us; up to a normalization factor, the eigenstates and energy eigenvalues are

$$\psi_m(x) = \sin\Big(m\pi x/L\Big), \quad E_m = \frac{1}{2}\left(\frac{m\pi}{L}\right)^2, \quad m = 1, 2, 3, \ldots \tag{8.4.3}$$

We will write a program that seeks a numerical solution. Using the three-point rule to discretize the second derivative, the finite-difference matrix equations become

$$-\frac{1}{2h^2}\begin{bmatrix} -2 & 1 & & \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{bmatrix}\begin{bmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{N-1} \end{bmatrix} = E\begin{bmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{N-1} \end{bmatrix}, \tag{8.4.4}$$

where

$$h = \frac{L}{N+1}, \quad \psi_n = \psi\Big(x = h(n+1)\Big) \tag{8.4.5}$$

The following program constructs the finite-difference matrix equation, and displays the first three solutions:

```
from scipy import *
import scipy.sparse as sp
import scipy.sparse.linalg as spl
import matplotlib.pyplot as plt

## Solve the 1D particle-in-a-box problem for box length L,
## using N discretization points.  The parameter nev is the
## number of eigenvalues/eigenvectors to find.  Return three
## arrays E, psi, and x.  E stores the energy eigenvalues;
## psi stores the (non-normalized) eigenstates; and x stores
## the discretization points.
def particle_in_a_box(L, N, nev=3):
    dx = L/(N+1.0)
    x  = linspace(dx, L-dx, N)
    I  = ones(N)
    ## Set up the finite-difference matrix.
    H  = sp.dia_matrix(([I, -2*I, I], [-1,0,1]), shape=(N,N))
    H *= -0.5/(dx*dx)
    ## Find the lowest eigenvalues and eigenvectors.
    E, psi = spl.eigsh(H, k=nev, sigma=-1.0)
    return E, psi, x
```

```
def particle_in_a_box_demo():
    E, psi, x = particle_in_a_box(1.0, 1000)

    ## Print the energy eigenvalues.
    print(E)

    ## Plot |psi(x)|^2 vs x for each solution found.
    fig = plt.figure()
    axs = plt.subplot(1,1,1)
    for n in range(len(E)):
        fig_label = "State #" + str(n)
        plt.plot(x, abs(psi[:,n])**2, label=fig_label, linewidth=2)
    plt.xlabel('x')
    plt.ylabel('|psi(x)|^2')

    ## Shrink the axis by 20%, so the legend can fit.
    box = axs.get_position()
    axs.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    ## Print the legend.
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    plt.show()

particle_in_a_box_demo()
```

The Hamiltonian matrix, which is tridiagonal, is constructed in the DIA sparse matrix format. The eigenvalues and eigenvectors are found with `eigsh`, which can be used because the Hamiltonian matrix is known to be Hermitian. Notice that we call `eigsh` using the `sigma` parameter, telling the eigensolver to find the eigenvalues closest in value to -1.0:

```
    E, psi = spl.eigsh(H, k=nev, sigma=-1.0)
```

This will find the lowest energy eigenvalues because, in this case, all energy eigenvalues are positive. (We use -1.0 instead of 0.0, because the algorithm does not work well when `sigma` is exactly zero.) If there is a negative potential present, we would have to find a different estimate for the lower bound of the energy eigenvalues, and pass that to `sigma`.

Alternatively, we could have called `eigsh` with an input `which='SA'`. This would tell the eigensolver to find the eigenvalue with the smallest value. We avoid doing this because the ARPACK eigensolver algorithm is relatively inefficient at finding small eigenvalues in `which` mode (and it can sometimes even fail to converge, if `k` is too small). Typically, if you are able to deduce a lower bound for the desired eigenvalues, it is preferable to use `sigma`.

Running the program prints the lowest energy eigenvalues:

```
[  4.93479815  19.73914399  44.41289171]
```

This agrees well with the analytical results $E_1 = \pi^2/2 = 4.934802$, $E_2 = 2\pi^2 = 19.739208$, and $E_3 = 9\pi^2/2 = 44.413219$. It also produces the plot shown below, which is likewise as expected.

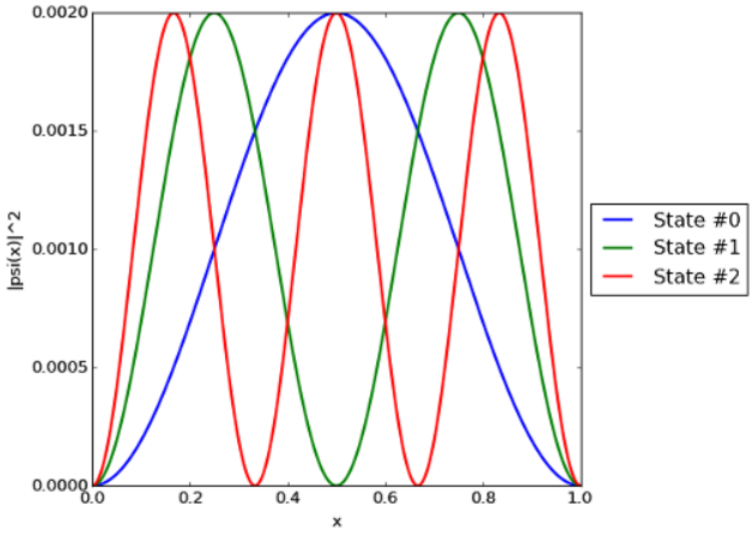

Figure 8.4.1: Plots of $|\psi(x)|^2$ versus x for the particle-in-a-box problem.

CHAPTER OVERVIEW

9: Numerical Integration

In this article, we will look at some basic techniques for numerically computing definite integrals. The most common techniques involve discretizing the integrals, which is conceptually similar to the way we discretized derivatives when studying finite-difference equations.

9.1: Mid-Point Rule

The simplest numerical integration method is called the **mid-point rule**. Consider a definite 1D integral

$$\mathcal{I} = \int_a^b f(x)\,dx. \tag{9.1.1}$$

Let us divide the range $a \le x \le b$ into a set of N segments of equal width, as shown in Fig. 9.1.1 for the case of $N = 5$. The mid-points of these segments are a set of N discrete points $\{x_0, \ldots x_{N-1}\}$, where

$$x_n = a + \left(n + \frac{1}{2}\right)\,\Delta x, \quad \Delta x \equiv \frac{b-a}{N}. \tag{9.1.2}$$

We then estimate the integral as

$$\mathcal{I}^{(\mathrm{mp})} = \Delta x \sum_{n=0}^{N-1} f(x_n) \quad \stackrel{N\to\infty}{\longrightarrow} \quad \mathcal{I}. \tag{9.1.3}$$

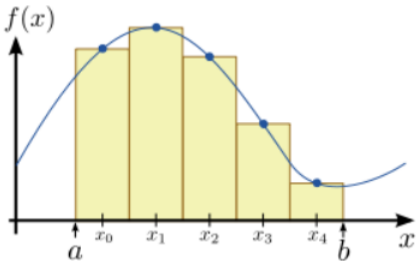

Figure 9.1.1: Computing a definite integral using the mid-point rule.

The principle behind this formula is very simple to understand. As shown in Fig. 9.1.1, I_N represents the area enclosed by a sequence of rectangles, where the height of each rectangle is equal to the value of $f(x)$ at its mid-point. As $N \to \infty$, the spacing between rectangles goes to zero; hence, the total area enclosed by the rectangles becomes equal to the area under the curve of $f(x)$.

9.1.1 Numerical Error for the Mid-Point Rule

Let us estimate the numerical error resulting from this approximation. To do this, consider one of the individual segments, which is centered at x_n with length $\Delta x = (b-a)/N$. Let us define the integral over this segment as

$$\Delta\mathcal{I}_n \equiv \int_{x_n - \Delta x/2}^{x_n + \Delta x/2} f(x) dx. \tag{9.1.4}$$

Now, consider the Taylor expansion of $f(x)$ in the vicinity of x_n:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2}(x-x_n)^2 + \frac{f'''(x_n)}{6}(x-x_n)^3 + \cdots \tag{9.1.5}$$

If we integrate both sides of this equation over the segment, the result is

$$\Delta\mathcal{I}_n = f(x_n)\Delta x + f'(x_n)\int_{x_n-\Delta x/2}^{x_n+\Delta x/2}(x-x_n)dx \tag{9.1.6}$$

$$+ \frac{f''(x_n)}{2}\int_{x_n-\Delta x/2}^{x_n+\Delta x/2}(x-x_n)^2 dx \tag{9.1.7}$$

$$+ \cdots \tag{9.1.8}$$

On the right hand side, every other term involves an integrand which is odd around x_n. Such terms integrate to zero. From the remaining terms, we find the following series for the integral of $f(x)$ over the segment:

$$\Delta\mathcal{I}_n = f(x_n)\Delta x + \frac{f''(x_n)\Delta x^3}{24} + O(\Delta x^5). \tag{9.1.9}$$

By comparison, the estimation provided by the mid-point rule is simply

$$\Delta \mathcal{I}_n^{\rm mp} = f(x_n)\Delta x \tag{9.1.10}$$

This is simply the first term in the exact series. The remaining terms constitute the numerical error in the mid-point rule integration, over this segment. We denote this error as

$$\mathcal{E}_n = |\Delta \mathcal{I}_n - \Delta \mathcal{I}_n^{\rm mp}| \ \sim \ \frac{|f''(x_n)|}{24}\Delta x^3 \ \sim \ O\left(\frac{1}{N^3}\right). \tag{9.1.11}$$

The last step comes about because, by our definition, $\Delta x \sim O(1/N)$.

Now, consider the integral over the entire integration range, which consists of N such segments. In general, there is no guarantee that the numerical errors of each segment will cancel out, so the total error should be N times the error from each segment. Hence, for the mid-point rule,

$$\mathcal{E}_{\rm total} \sim O\left(\frac{1}{N^2}\right). \tag{9.1.12}$$

9.2: Trapezium Rule

The **trapezium rule** is a simple extension of the mid-point integration rule. Instead of calculating the area of a sequence of rectangles, we calculate the area of a sequence of trapeziums, as shown in Fig. 9.2.1. As before, we divide the integration region into N equal segments; however, we now consider the *end-points* of these segments, rather than their mid-points. These is a set of $N+1$ positions $\{x_0, \ldots x_N\}$, such that

$$x_n = a + n\,\Delta x, \quad \Delta x \equiv \frac{b-a}{N}. \tag{9.2.1}$$

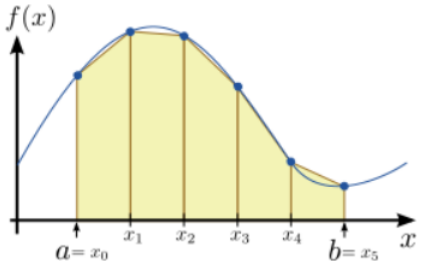

Figure 9.2.1: Computing a definite integral using the trapezium rule.

Note that $a = x_0$ and $b = x_N$. By using the familiar formula for the area of each trapezium, we can approximate the integral as

$$\mathcal{I}^{\text{trapz}} = \sum_{n=0}^{N-1} \Delta x \left(\frac{f(x_n) + f(x_{n+1})}{2} \right) \tag{9.2.2}$$

$$= \Delta x \left[\left(\frac{f(x_0) + f(x_1)}{2} \right) + \left(\frac{f(x_1) + f(x_2)}{2} \right) + \cdots + \left(\frac{f(x_{N-1}) + f(x_N)}{2} \right) \right] \tag{9.2.3}$$

$$= \Delta x \left[\frac{f(x_0)}{2} + \left(\sum_{n=1}^{N-1} f(x_n) \right) + \frac{f(x_N)}{2} \right]. \tag{9.2.4}$$

9.2.1 Python Implementation of the Trapezium Rule

In Scipy, the trapezium rule is implemented by the `trapz` function. It usually takes two array arguments, `y` and `x`; then the function call `trapz(y,x)` returns the trapezium rule estimate for $\int y\,dx$, using the elements of `x` as the discretization points, and the elements of `y` as the values of the integrand at those points.

Note

Matlab compatibility note: Be careful of the order of inputs! For the Scipy function, it's `trapz(y,x)`. For the Matlab function of the same name, the inputs are supplied in the opposite order, as `trapz(x,y)`.

Note that the discretization points in `X` need not be equally-spaced. Alternatively, you can call the function as `trapz(y,dx=s)`; this performs the numerical integration assuming equally-spaced discretization points, with spacing `s`.

Here is an example of using `trapz` to evaluate the integral $\int_0^\infty \exp(-x^2)dx = 1$:

```
>>> from scipy import *
>>> x = linspace(0,10,25)
>>> y = exp(-x)
>>> t = trapz(y,x)
>>> print(t)
1.01437984777
```

9.2.2 Numerical Error for the Trapezium Rule

From a visual comparison of Fig. 9.1.1 (for the mid-point rule) and Fig. 9.2.1 (for the trapezium rule), we might be tempted to conclude that the trapezium rule should be more accurate. Before jumping to that conclusion, however, let's do the actual calculation of the numerical error. This is similar to our above calculation of the mid-point rule's numerical error. For the trapezium rule, however, it's convenient to consider a *pair* of adjacent segments, which lie between the three discretization points $\{x_{n-1}, x_n, x_{n+1}\}$.

As before, we perform a Taylor expansion of $f(x)$ in the vicinity of x_n:

$$f(x) = f(x_n) + f'(x_n)(x-x_n) + \frac{f''(x_n)}{2}(x-x_n)^2 + \frac{f'''(x_n)}{6}(x-x_n)^3 + \cdots \tag{9.2.5}$$

If we integrate $f(x)$ from x_{n-1} to x_{n+1}, the result is

$$\mathcal{I}_n \equiv \int_{x_n-\Delta x}^{x_n+\Delta x} f(x)dx \tag{9.2.6}$$

$$= 2f(x_n)\Delta x \; + \; f'(x_n)\int_{x_n-\Delta x}^{x_n+\Delta x}(x-x_n)dx \; + \; \frac{f''(x_n)}{2}\int_{x_n-\Delta x}^{x_n+\Delta x}(x-x_n)^2 dx \; + \cdots \tag{9.2.7}$$

As before, every other term on the right-hand side integrates to zero. We are left with

$$\mathcal{I}_n \; = \; 2f(x_n)\Delta x \; + \; \frac{f''(x_n)}{3}\Delta x^3 + O(\Delta x^5) \; + \cdots \tag{9.2.8}$$

This has to be compared to the estimate produced by the trapezium rule, which is

$$\mathcal{I}_n^{(\text{trapz})} = \Delta x\left[\frac{f(x_{n-1})}{2} + f(x_n) + \frac{f(x_{n+1})}{2}\right]. \tag{9.2.9}$$

If we Taylor expand the first and last terms of the above expression around x_n, the result is:

$$\mathcal{I}_n^{(\text{trapz})} \; = \frac{\Delta x}{2}\left[f(x_n) - f'(x_n)\Delta x + \frac{f''(x_n)}{2}\Delta x^2 - \frac{f'''(x_n)}{6}\Delta x^3 + \frac{f''''(x_n)}{24}\Delta x^4 + O(\Delta x^4)\right] \tag{9.2.10}$$

$$+ \; \Delta x \; f(x_n) \tag{9.2.11}$$

$$+ \frac{\Delta x}{2}\left[f(x_n) + f'(x_n)\Delta x + \frac{f''(x_n)}{2}\Delta x^2 + \frac{f'''(x_n)}{6}\Delta x^3 + \frac{f''''(x_n)}{24}\Delta x^4 + O(\Delta x^4)\right] \tag{9.2.12}$$

$$= 2f(x_n)\Delta x + \frac{f''(x_n)}{2}\Delta x^3 + O(\Delta x^5). \tag{9.2.13}$$

Hence, the numerical error for integrating over this pair of segments is

$$\mathcal{E}_n \; \equiv \; |\mathcal{I}_n - \mathcal{I}_n^{\text{trapz}}| \; = \; \frac{|f''(x_n)|}{6}\Delta x^3 \; \sim \; O\left(\frac{1}{N^3}\right). \tag{9.2.14}$$

And the total numerical error goes as

$$\mathcal{E}_{\text{total}} \sim O\left(\frac{1}{N^2}\right), \tag{9.2.15}$$

which is the same scaling as the mid-point rule! Despite our expectations, the trapezium rule isn't actually an improvement over the mid-point rule, as far as numerical accuracy is concerned.

9.3: Simpson's Rule

Our analysis of the mid-point rule and the trapezium rule showed that both methods have $O(1/N^2)$ numerical error. In both cases, the error can be traced to the same source: the fact that the integral estimate over each segment differs from the Taylor series result in the *second-order* term—the term proportional to $f''(x_n)$. This suggests a way to improve on the numerical integration result: we could take a weighted average of the mid-point rule and the trapezium rule, such that the *second-order* numerical errors from the two schemes cancel each other out! This is the numerical integration method known as **Simpson's rule**.

To be precise, let's again consider a pair of adjacent segments, which lie between the equally-spaced discretization points $\{x_{n-1}, x_n, x_{n+1}\}$. As derived above, the integral over these segments can be Taylor expanded as

$$\mathcal{I}_n = 2f(x_n)\Delta x + \frac{f''(x_n)}{3}\Delta x^3 + O(\Delta x^5) + \cdots \tag{9.3.1}$$

By comparison, the mid-point rule and trapeium rule estimators for the integral are

$$\mathcal{I}_n^{\text{mp}} = 2f(x_n)\Delta x \tag{9.3.2}$$

$$\mathcal{I}_n^{\text{trapz}} = 2f(x_n)\Delta x + \frac{f''(x_n)}{2}\Delta x^3 + O(\Delta x^5). \tag{9.3.3}$$

Hence, we could take the following weighted average:

$$\mathcal{I}_n^{\text{simp}} = \frac{1}{3}\mathcal{I}_n^{\text{mp}} + \frac{2}{3}\mathcal{I}_n^{\text{trapz}} = 2f(x_n)\Delta x + \frac{f''(x_n)}{3}\Delta x^3 + O(\Delta x^5). \tag{9.3.4}$$

Such a weighted average would match the Taylor series result up to $O(\Delta x^4)$! (You can check for yourself that the $O(\Delta x^5)$ terms differ.) In summary, Simpson's rule for this set of three points can be written as

$$\mathcal{I}_n^{\text{simp}} = \frac{1}{3}\Big[2f(x_n)\Delta x\Big] + \frac{2}{3}\,\Delta x\Big[\frac{f(x_{n-1})}{2} + f(x_n) + \frac{f(x_{n+1})}{2}\Big] \tag{9.3.5}$$

$$= \frac{\Delta x}{3}\Big[f(x_{n-1}) + 4f(x_n) + f(x_{n+1})\Big]. \tag{9.3.6}$$

The total numerical error, over a set of $O(N)$ segments, is $O(1/N^4)$. That is an improvement of two powers of $1/N$ over the trapzezium and mid-point rules! What's even better is that it involves exactly the same number of arithmetic operations as the trapezium rule. This is as close to a free lunch as you can get in computational science.

9.3.1 Python Implementation of Simpson's Rule

In Scipy, Simpson's rule is implemented by the `scipy.integrate.simps` function, which is defined in the `scipy.integrate` submodule. Similar to the `trapz` function, this can be called as either `simps(y,x)` or `simps(y,dx=s)` to estimate the integral $\int y\,dx$, using the elements of `x` as the discretization points, with `y` specifying the set of values for the integrand.

Because Simpson's rule requires dividing the segments into pairs, if you specify an even number of discretization points in `x` (i.e. an odd number of segments), the function deals with this by doing a trapezium rule estimate on the first and last segments. Usually, the error is negligible, so don't worry about this detail

Here is an example of `simps` in action:

```
>>> from scipy import *
>>> from scipy.integrate import simps
>>> x = linspace(0,10,25)
>>> y = exp(-x)
>>> t = simps(y,x)
>>> print(t)
1.00011864276
```

For the same number of discretization points, the trapezium rule gives 1.01438; the exact result is $0.9999546\ldots$ Clearly, Simpson's rule is more accurate.

9.4: Gaussian Quadratures

Previously, we have assumed in our analysis of numerical integration schemes that the discretization points are equally-spaced. This assumption is not strictly necessary; for instance, we can easily modify the mid-point rule and trapezium rule formulas to work with non-equally-spaced points.

However, if we are free to choose the discretization points for performing numerical integration, and these points need not be equally spaced, then it is possible to exploit this freedom to further improve the accuracy of the numerical integration. The idea is to optimize the placement of the discretization points, so as to minimize the resulting numerical error. This is the basic idea behind the method of integration by **Gaussian quadratures**.

We will not discuss the details of this numerical integration method. To use it, you can call the `scipy.integrate.quad` function. This function uses a low-level numerical library named QUADPACK, which performs quadrature integration with **adaptive quadratures**—i.e., it automatically figures out how many discretization points should be used, and where they should be located, in order to produce a result with the desired numericaly accuracy.

Because QUADPACK figures out the discretization points for itself, you have to pass `quad` a *function* representing the integrand, rather than an array of integrand values as with `trapz` or `simps`. The standard way to call the function is

```
t = quad(f, a, b)
```

which calculates the integral

$$t = \int_a^b f(x)dx. \tag{9.4.1}$$

The return value is a tuple of the form `(t,err)`, where `t` is the value of the integral and `err` is an estimate of the numerical error. The `quad` function also accepts many other optional inputs, which can be used to specify additional inputs for passing to the integrand function, the error tolerance, the number of subintervals to use, etc. See the documentation for details.

The choice of whether to perform numerical integration using Simpson's rule (`simps`) or Gaussian quadratures (`quad`) is situational. If you already know the values of the integrands at a pre-determined set of discretization points (e.g., from the result of a finite-difference calculation), then use `simps`. If you can define a function that can quickly calculate the value of the integrand at any point, use `quad`.

Here is an example of using `quad` to compute the integral $\int_0^\infty \frac{dx}{x^2+1}$:

```
from scipy import *
from scipy.integrate import quad

def f(x):
    return 1.0 / (x*x+1)

integ, _ = quad(f, 0, 1000)
print(integ)
```

(Note that `quad` returns two values; the first is the computed value of the integral, and the other is the absolute error, which we're not interested in, so we toss it in the "throwaway" variable named `_`. See the documentation for details.) Running the above program prints the result $1.569796\ldots$,which differs from the exact result, $\pi/2 = 1.570796\ldots$,by a relative error of 0.06%.

Here is another example, where the integrand takes an additional parameter: $\int_0^\infty xe^{-\lambda x}\,dx$:

```
from scipy import *
from scipy.integrate import quad
```

```
def f(x, lambd):
    return x * exp(-lambd * x)

integ, _ = quad(f, 0, 100, args=(0.5))
print(integ)
```

Running the program prints the result 4.0, which agrees with the exact result of $1/\lambda^2$ for the chosen value of the parameter $\lambda = 0.5$.

9.5: Monte Carlo Integration

The final numerical integration scheme that we'll discuss is **Monte Carlo integration**, and it is conceptually completely different from the previous schemes. Instead of assigning a set of discretization points (either explicitly, as in the mid-point/trapezium/Simpson's rules, or through a machine optimization procedure, as in the adaptive quadrature method), this method *randomly* samples the points in the integration domain. If the sampling points are independent and there is a sufficiently large number of them, the integral can be estimated by taking a weighted average of the integrand over the sampling points.

To be precise, consider a 1D integral over a domain $x \in [a, b]$. Let each sampling point be drawn independently from a distribution $p(x)$. This means that the probability of drawing sample x_n in the range $x_n \in [x, x+dx]$ is $p(x)dx$. The distribution is normalized, so that

$$\int_a^b p(x)\, dx = 1. \tag{9.5.1}$$

Let us take N samples, and evaluate the integrand at those points: this gives us a set of numbers $\{f(x_n)\}$. We then compute the quantity

$$\mathcal{I}^{\rm mc} = \frac{1}{N} \sum_{n=0}^{N-1} \frac{f(x_n)}{p(x_n)}. \tag{9.5.2}$$

Unlike the estimators that we have previously studied, $\mathcal{I}^{\rm mc}$ is a random number (because the underlying variables $\{x_n\}$ are all random). Crucially, its *average* value is equal to the desired integral:

$$\begin{aligned}
\langle \mathcal{I}^{\rm mc} \rangle &= \frac{1}{N} \sum_{n=0}^{N-1} \left\langle \frac{f(x_n)}{p(x_n)} \right\rangle && (9.5.3)\\
&= \left\langle \frac{f(x_n)}{p(x_n)} \right\rangle \qquad \text{for each } n && (9.5.4)\\
&= \int_a^b p(x) \left[\frac{f(x)}{p(x)} \right] dx && (9.5.5)\\
&= \int_a^b f(x)\, dx && (9.5.6)
\end{aligned}$$

For low-dimensional integrals, there is normally no reason to use the Monte Carlo integration method. It requires a much larger number of samples in order to reach a level of numerical accuracy comparable to the other numerical integration methods. (For 1D integrals, Monte Carlo integration typically requires millions of samples, whereas Simpson's rule only requires hundreds or thousands of discretization points.) However, Monte Carlo integration outperforms discretization-based integration schemes when the dimensionality of the integration becomes extremely large. Such integrals occur, for example, in quantum mechanical calculations involving many-body systems, where the dimensionality of the Hilbert space scales exponentially with the number of particles.

CHAPTER OVERVIEW

10: Numerical Integration of ODEs

This article describes the numerical methods for solving the **initial-value problem**, which is a standard type of problem appearing in many fields of physics. Suppose we have a system whose state at time t is described by a vector $\vec{y}(t)$, which obeys the first-order ordinary differential equation (ODE) for the form:

$$\frac{d\vec{y}}{dt} = \vec{F}\Big(\vec{y}(t), t\Big). \tag{10.1}$$

Here, $\vec{F}$ is some given vector-valued function, whose inputs are (i) the instantaneous state $\vec{y}(t)$ and (ii) the current time t. Then, given an initial time t_0 and an initial state $\vec{y}(t_0)$, the goal is to find $\vec{y}(t)$ for subsequent times.

Conceptually, the initial value problem is distinct from the problem of solving an ODE discussed in the article on finite-difference equations. There, we were given a pair of boundaries with certain boundary conditions, and the goal was to find the solution between the two boundaries. In this case, we are given the state at an initial time t_0, and our goal is to find $\vec{y}(t)$ for some set of future times $t > t_0$. This is sometimes referred to as "integrating" the ODE, because the solution has the form

$$\vec{y}(t) = \vec{y}(t_0) + \int_{t_0}^{t} dt' \ \vec{F}\Big(\vec{y}(t'), t'\Big). \tag{10.2}$$

However, unlike ordinary numerical integration (i.e., the computing of a definite integral), the value of the integrand is not known in advance, because of the dependence of $\vec{F}$ on the unknown $\vec{y}(t)$.

10.1: Example- Equations of Motion in Classical Mechanics

The above standard formulation of the initial-value problem can be used to describe a very large class of time-dependent ODEs found in physics. For example, suppose we have a classical mechanical particle with position $\vec{r}$, subject to an arbitrary external space-and-time-dependent force $\vec{f}\,(\vec{r},t)$ and a friction force $-\lambda d\vec{r}/dt$ (where λ is a damping coefficient). Newton's second law gives the following equation of motion:

$$m\frac{d^2\vec{r}}{dt^2} = -\lambda\frac{d\vec{r}}{dt} + \vec{f}\,(\vec{r},t). \tag{10.1.1}$$

This is a second-order ODE, whereas the standard initial-value problem involves a first-order ODE. However, we can turn it into a first-order ODE with the following trick. Define the velocity vector

$$\vec{v} = \frac{d\vec{r}}{dt}, \tag{10.1.2}$$

and define the state vector by combining the position and velocity vectors:

$$\vec{y} = \begin{bmatrix} \vec{r} \\ \vec{v} \end{bmatrix}. \tag{10.1.3}$$

Then the equation of motion takes the form

$$\frac{d\vec{y}}{dt} = \frac{d}{dt}\begin{bmatrix} \vec{r} \\ \vec{v} \end{bmatrix} = \begin{bmatrix} \vec{v} \\ -(\lambda/m)\vec{v} + \vec{f}\,(\vec{r},t)/m \end{bmatrix}, \tag{10.1.4}$$

which is a first-order ODE, as desired. The quantity on the right-hand side is the derivative function $\vec{F}(\vec{y},t)$ for the initial-value problem. Its dependence on $\vec{r}$ and $\vec{v}$ is simply regarded as a dependence on the upper and lower portions of the state vector $\vec{y}$. In particular, note that the derivative function does not need to be linear, since $\vec{f}$ can have any arbitrary nonlinear dependence on $\vec{r}$, e.g. it could depend on the quantity $|\vec{r}|$.

The "initial state", $\vec{y}(t_0)$, requires us to specify both the initial position and velocity of the particle, which is consistent with the fact that the original equation of motion was a second-order equation, requiring two sets of initial values to fully specify a solution. In a similar manner, ODEs of higher order can be converted into first-order form, by defining the higher derivatives as state variables and increasing the size of the state vector.

10.2: Forward Euler Method

The **Forward Euler Method** is the conceptually simplest method for solving the initial-value problem. For simplicity, let us discretize time, with equal spacings:

$$t_0, t_1, t_2, \ldots \quad \text{where} \quad h \equiv t_{n+1} - t_n. \tag{10.2.1}$$

Let us denote $\vec{y}_n \equiv \vec{y}(t_n)$. The Forward Euler Method consists of the approximation

$$\vec{y}_{n+1} = \vec{y}_n + h\vec{F}(\vec{y}_n, t_n). \tag{10.2.2}$$

Starting from the initial state $\vec{y}_0$ and initial time t_0, we apply this formula repeatedly to compute $\vec{y}_1, \vec{y}_2$, and so forth. The Forward Euler Method is called an **explicit method**, because, at each step n, all the information that you need to calculate the state at the next time step, $\vec{y}_{n+1}$, is already explicitly known—i.e., you just need to plug $\vec{y}_n$ and t_n into the right-hand side of the above formula.

The Forward Euler Method formula follows from the usual definition of the derivative, and becomes exactly correct as $h \to 0$. We can deduce the numerical error, which is called the **local truncation error** in this context, by Taylor expanding the left-hand side around $t = t_n$:

$$\vec{y}_{n+1} = \vec{y}_n + h\left.\frac{d\vec{y}}{dt}\right|_{t_n} + \frac{h^2}{2}\left.\frac{d^2\vec{y}}{dt^2}\right|_{t_n} + \cdots \tag{10.2.3}$$

The first two terms are precisely equal to the right-hand side of the Forward Euler Method formula. The local truncation error is the magnitude of the remaining terms, and hence it scales as $O(h^2)$.

10.2.1 Instability

For the Forward Euler Method, the local truncation error leads to a profound problem known as **instability**. Because the method involves repeatedly applying a formula with a local truncation error at each step, it is possible for the errors on successive steps to progressively accumulate, until the solution itself blows up. To see this, consider the differential equation

$$\frac{dy}{dt} = -\kappa y. \tag{10.2.4}$$

Given an initial state y_0 at time $t_0 = 0$, the solution is $y(t) = y_0\, e^{-\kappa t}$. For $\kappa > 0$, this decays exponentially to zero with increasing time. However, consider the solutions produced by the Forward Euler Method:

$$y_1 = y_0 + h\cdot(-\kappa y_0) = (1 - h\kappa)\, y_0 \tag{10.2.5}$$
$$y_2 = y_1 + h\cdot(-\kappa y_1) = (1 - h\kappa)^2\, y_0 \tag{10.2.6}$$
$$\vdots = \vdots \tag{10.2.7}$$
$$y_n = (1 - h\kappa)^n\, y_0. \tag{10.2.8}$$

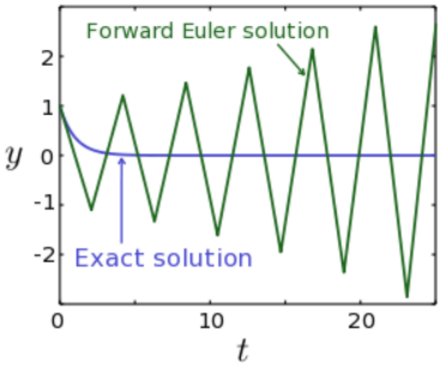

Figure 10.2.1: The exact solution (blue) and Forward Euler solution (green) for $dy/dt = -\kappa y(t)$, for $y(0) = 1$, $\kappa = 1$, and $h = 2.1$. The numerical solution is unstable; it blows up at large times, even though the exact solution is decaying to zero.

If $h > 2/\kappa$, then $1 - h\kappa < -1$, and as a result $y_n \to \pm\infty$ as $n \to \infty$. Even though the actual solution decays exponentially to zero, the numerical solution blows up, as shown in Fig. 10.2.1. Roughly speaking, the local truncation error causes the numerical

solution to "overshoot" the true solution; if the step size is too large, the magnitude of the overshoot keeps growing with each step, destabilizing the numerical solution.

Stability is a fundamental problem for the integration of ODEs. The equations which tend to destabilize numerical ODE solvers are those containing spring constants which are "large" compared to the time step; such equations are called **stiff equations**. At first, you might think that it's no big deal: just make the step size h sufficiently small, and the blow-up can be avoided. The trouble is that it's often unclear how small is sufficiently small, particularly for complicated (e.g. nonlinear) ODEs, where $F(\vec{y}, t)$ is something like a "black box". Unlike the above simple example, we typically don't have the exact answer to compare with, so it can be hard to tell whether the numerical solution blows up because that's how the true solution behaves, or because the numerical method is unstable.

10.3: Backward Euler Method

In the **Backward Euler Method**, we take

$$\vec{y}_{n+1} = \vec{y}_n + h\vec{F}(\vec{y}_{n+1}, t_{n+1}). \tag{10.3.1}$$

Comparing this to the formula for the Forward Euler Method, we see that the inputs to the derivative function involve the solution at step $n+1$, rather than the solution at step n. As $h \to 0$, both methods clearly reach the same limit. Similar to the Forward Euler Method, the local truncation error is $O(h^2)$.

Because the quantity $\vec{y}_{n+1}$ appears in both the left- and right-hand sides of the above equation, the Backward Euler Method is said to be an **implicit method** (as opposed to the Forward Euler Method, which is an explicit method). For general derivative functions F, the solution for $\vec{y}_{n+1}$ cannot be found directly, but has to be obtained *iteratively*, using a numerical approximation technique such as Newton's method. This makes the Backward Euler Method substantially more complicated to implement, and slower to run.

However, implicit methods like the Backward Euler Method have a powerful advantage: it turns out that they are generally stable regardless of step size. By contrast, explicit methods—even explicit methods that are much more sophisticated than the Forward Euler Method, like the Runge-Kutta methods discussed below—are unstable when applied to stiff problems, if the step size is too large. To illustrate this, let us apply the Backward Euler Method to the same ODE, $dy/dt = -\kappa y(t)$, discussed previously. For this particular ODE, the implicit equation can be solved exactly, without having to use an iterative solver:

$$y_{n+1} = y_n - h\kappa y_{n+1} \quad \Rightarrow \quad y_{n+1} = \frac{y_n}{1+h\kappa} = \frac{y_0}{(1+h\kappa)^n}. \tag{10.3.2}$$

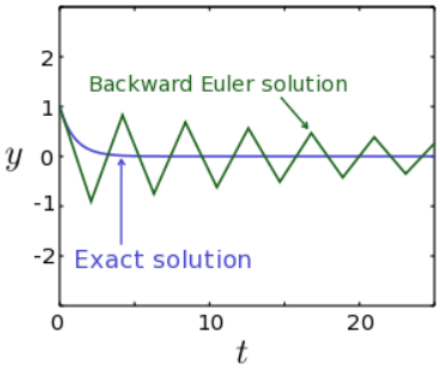

Figure 10.3.1: The exact solution (blue) and Backward Euler solution (green) for the same problem as Fig. 10.2.1. The numerical solution is stable.

From this result, we can see that the numerical solution does not blow up for large values of h, as shown for example in Fig. 10.3.1. Even though the numerical solution in this example isn't accurate (because of the large value of h), the key point is that the error does not accumulate and cause a blow-up at large times.

10.4: Adams-Moulton Method

In the **Second-Order Adams-Moulton** (AM2) method, we take

$$\vec{y}_{n+1} = \vec{y}_n + \frac{h}{2}\left[\vec{F}(\vec{y}_n, t_n) + \vec{F}(\vec{y}_{n+1}, t_{n+1})\right]. \tag{10.4.1}$$

Conceptually, the derivative term here is the average of the Forward Euler and Backward Euler derivative terms. Because $\vec{y}_{n+1}$ appears on the right-hand side, this is an implicit method. Thus, like the Backward Euler Method, it typically has to be solved iteratively, but is numerically stable. The advantage of the AM2 method is that its local truncation error is substantially lower. To see this, let us take the derivative of both sides of the ODE over one time step:

$$\int_{t_n}^{t_{n+1}} \frac{d\vec{y}}{dt}\, dt = \int_{t_n}^{t_{n+1}} \vec{F}(\vec{y}(t), t)\, dt \tag{10.4.2}$$

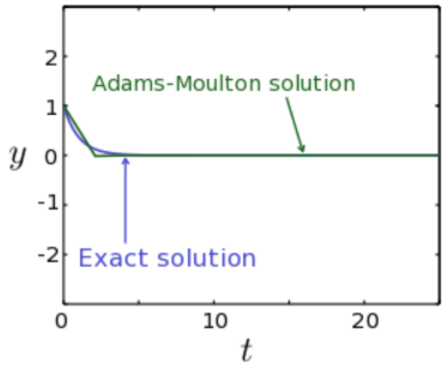

Figure 10.4.1: The exact solution (blue) and Second-Order Adams-Moulton (AM2) solution (green) for the same problem as Fig. 10.2.1.

The integral on the left-hand side reduces to $\vec{y}_{n+1} - \vec{y}_n$. As for the integral on the right-hand side, if we perform this integral numerically using the trapezium rule, then the result is the derivative term in the AM2 formula. The local truncation error is given by the numerical error of the trapezium rule, which is $O(h^3)$. That's an improvement of one order compared to the Euler methods. (Based on this argument, we can also see that the Forward Euler method and the Backward Euler methods involve approximating the integral on the right-hand side using a rectangular area, with height given by the value at t_n and t_{n+1} respectively. From this, it's clear why the AM2 scheme gives better results.)

There are also higher-order Adams-Moulton methods, which generate even more accurate results by also sampling the derivative function at previous steps: $F(\vec{y}_{n-1}, t_{n-1})$, $F(\vec{y}_{n-2}, t_{n-2})$, etc.

In Fig. 10.4.1, we plot the AM2 solution for the problem $dy/dt = -\kappa y(t)$, using the same parameters (including the same step size h) as in Fig. 10.2.1 (Forward Euler Method) and Fig. 10.3.1 (Backward Euler Method). It is clear that the AM2 results are significantly more accurate.

10.5: Runge-Kutta Methods

The three methods that we have surveyed thus far (Forward Euler, Backward Euler, and Adams-Moulton) have all involved sampling the derivative function $F(y,t)$ at one of the discrete time steps $\{t_n\}$, and the solutions at those time steps $\{\vec{y}_n\}$. It is natural to ask whether we can improve the accuracy by sampling the derivative function at "intermediate" values of t and $\vec{y}$. This is the basic idea behind a family of numerical methods known as **Runge-Kutta** methods.

Here is a simple version of the method, known as **second-order Runge-Kutta** (RK2). Our goal is to replace the derivative term with a pair of terms, of the form

$$\vec{y}_{n+1} = \vec{y}_n + Ah\vec{F}_A + Bh\vec{F}_B, \tag{10.5.1}$$

where

$$\vec{F}_A = \vec{F}(\vec{y}_n, t_n) \tag{10.5.2}$$
$$\vec{F}_B = \vec{F}(\vec{y}_n + \beta\vec{F}_A, t_n + \alpha). \tag{10.5.3}$$

The coefficients $\{A, B, \alpha, \beta\}$ are adjustable parameters whose values we'll shortly choose, so as to minimize the local truncation error.

During each time step, we start out knowing the solution $\vec{y}_n$ at time t_n, we first calculate $\vec{F}_A$ (which is the derivative term that goes into the Forward Euler method); then we use that to calculate an "intermediate" derivative term $\vec{F}_B$. Finally, we use a weighted average of $\vec{F}_A$ and $\vec{F}_B$ as the derivative term for calculating $\vec{y}_{n+1}$. From this, it is evident that this is an *explicit* method: for each of the sub-equations, the "right-hand sides" contain known quantities.

We now have to determine the appropriate values of the parameters $\{A, B, \alpha, \beta\}$. First, we Taylor expand $\vec{y}_{n+1}$ around t_n, using the chain rule:

$$\vec{y}_{n+1} = \vec{y}_n + h\frac{d\vec{y}}{dt}\bigg|_{t_n} + \frac{h^2}{2}\frac{d^2\vec{y}}{dt^2}\bigg|_{t_n} + O(h^3) \tag{10.5.4}$$
$$= \vec{y}_n + h\vec{F}(\vec{y}_n, t_n) + \frac{h^2}{2}\left[\frac{d}{dt}\vec{F}(\vec{y}(t), t)\right]_{t_n} + O(h^3) \tag{10.5.5}$$
$$= \vec{y}_n + h\vec{F}(\vec{y}_n, t_n) + \frac{h^2}{2}\left[\sum_j \frac{\partial\vec{F}}{\partial y_j}\frac{dy_j}{dt} + \frac{\partial\vec{F}}{\partial t}\right]_{t_n} + O(h^3) \tag{10.5.6}$$
$$= \vec{y}_n + h\vec{F}_A + \frac{h^2}{2}\left\{\sum_j \left[\frac{\partial\vec{F}}{\partial y_j}\right]_{t_n} F_{Aj} + \left[\frac{\partial\vec{F}}{\partial t}\right]_{t_n}\right\} + O(h^3) \tag{10.5.7}$$

In the same way, we Taylor expand the intermediate derivative term F_B, whose formula was given above:

$$F_B = F_A + \beta\sum_j F_{Aj}\left[\frac{\partial F}{\partial y_j}\right]_{t_n} + \alpha\left[\frac{\partial F}{\partial t}\right]_{t_n}. \tag{10.5.8}$$

If we compare these Taylor expansions to the RK2 formula, then it can be seen that the terms can be made to match up to (and including) $O(h^2)$, if the parameters are chosen to obey the equations

$$A + B = 1, \quad \alpha = \beta = \frac{h}{2B}. \tag{10.5.9}$$

One possible set of solutions is $A = B = 1/2$ and $\alpha = \beta = h$. With these conditions met, the RK2 method has local truncation error of $O(h^3)$, one order better than the Forward Euler Method (which is likewise an explicit method), and comparable to the Adams-Moulton Method (which is an implicit method).

The local truncation error can be further reduced by taking more intermediate samples of the derivative function. The most commonly-used Runge-Kutta method is the **fourth-order Runge Kutta method** (RK4), which is given by

$$\vec{y}_{n+1} = \vec{y}_n + \frac{h}{6}\left(\vec{F}_A + 2\vec{F}_B + 2\vec{F}_C + \vec{F}_D\right) \tag{10.5.10}$$

$$\vec{F}_A = \vec{F}(\vec{y}_n,\, t_n), \tag{10.5.11}$$

$$\vec{F}_B = \vec{F}(\vec{y}_n + \tfrac{h}{2}\vec{F}_A,\, t_n + \tfrac{h}{2}), \tag{10.5.12}$$

$$\vec{F}_C = \vec{F}(\vec{y}_n + \tfrac{h}{2}\vec{F}_B,\, t_n + \tfrac{h}{2}), \tag{10.5.13}$$

$$\vec{F}_D = \vec{F}(y_n + h\vec{F}_C,\, t_n + h). \tag{10.5.14}$$

This has local truncation error of $O(h^5)$. It is an explicit method, and therefore has the disadvantage of being unstable if the problem is stiff and h is sufficiently large.

10.6: Integrating ODEs with Scipy

Except for educational purposes, it is almost always a bad idea to implement your own ODE solver; instead, you should use a pre-written solver.

10.6.1 The `scipy.integrate.odeint` Solver

In Scipy, the simplest ODE solver to use is the `scipy.integrate.odeint` function, which is in the `scipy.integrate` module. This is actually a wrapper around a low-level numerical library known as LSODE (the **L**ivermore **S**olver for **ODE**s"), which is part of a widely-used ODE solver library known as ODEPACK. The most important feature of this solver is that it is "adaptive": it can automatically figure out (i) which integration scheme to use (choosing between either a high-order Adams-Moulton method, or another implicit method known as the Backward Differentiation Formula which we haven't described), and (ii) the size of the discrete time steps, based on the behavior of the solutions as they are being worked out. In other words, the user only needs to specify the derivative function, the initial state, and the desired output times, without having to worry about the internal details of the solution method.

The function takes several inputs, of which the most important ones are:

1. `func`, a function corresponding to the derivative function $\vec{F}(\vec{y}, t)$.
2. `y0`, either a number or 1D array, corresponding to the initial state $\vec{y}(t_0)$.
3. `t`, an array of times at which to output the ODE solution. The first element corresponding to the initial time t_0. Note that these are the "output" times only—they do *not* specify the actual time steps which the solver uses for finding the solutions; those are automatically determined by the solver.
4. (optional) `args`, a tuple of extra inputs to pass to the derivative function `func`. For example, if `args=(2,3)`, then `func` should accept four inputs, and it will be passed 2 and 3 as the last two inputs.

The function then returns an array `y`, where `y[n]` contains the solution at time `t[n]`. Note that `y[0]` will be exactly the same as the input `y0`, the initial state which you specified.

Here is an example of using `odeint` to solve the damped harmonic oscillator problem $m\ddot{x} = -\lambda\dot{x} - kx(t)$, using the previously-mentioned vectorization trick to cast it into a first-order ODE:

```
from scipy import *
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def ydot(y, t, m, lambd, k):
    x, v = y[0], y[1]
    return array([v, -(lambd/m) * v - k * x / m])

m, lambd, k = 1.0, 0.1, 1.0   # Oscillator parameters
y0 = array([1.0, 5.0])        # Initial conditions [x, v]
t  = linspace(0.0, 50.0, 100) # Output times

y = odeint(ydot, y0, t, args=(m, lambd, k))

## Plot x versus t
plt.plot(t, y[:,0], 'b-')
plt.xlabel('t')
plt.ylabel('x')
plt.show()
```

There is an important limitation of `odeint` : it **does not handle complex ODEs**, and always assumes that $\vec{y}$ and $\vec{F}$ are real. However, this is not a problem in practice, because you can always convert a complex first-order ODE into a real one, by replacing the complex vectors $\vec{y}$ and $\vec{F}$ with double-length real vectors:

$$\vec{y}' \equiv \begin{bmatrix} \mathrm{Re}(\vec{y}) \\ \mathrm{Im}(\vec{y}) \end{bmatrix}, \quad \vec{F}' \equiv \begin{bmatrix} \mathrm{Re}(\vec{F}) \\ \mathrm{Im}(\vec{F}) \end{bmatrix}. \tag{10.6.1}$$

10.6.2 The `scipy.integrate.ode` Solvers

Apart from `odeint` , Scipy provides a more general interface to a variety of ODE solvers, in the form of the `scipy.integrate.ode` class. This is a much more low-level interface; instead of calling a single function, you have to create an ODE "object", then use the methods of this object to specify the type of ODE solver to use, the initial conditions, etc.; then you have to repeatedly call the ODE object's `integrate` method, to integrate the solution up to each desired output time step.

The is an extremely aggravating inconsistency between the `odeint` function and this `ode` class: the expected order of inputs for the derivative functions are reversed! The `odeint` function assumes the derivative function has the form `F(y,t)` , but the `ode` class assumes it has the form `F(t,y)` . Watch out for this!

Here is an example of using `ode` class with the damped harmonic oscillator problem $m\ddot{x} = -\lambda\dot{x} - kx(t)$, using a Runge-Kutta solver:

```
from scipy import *
import matplotlib.pyplot as plt
from scipy.integrate import ode

## Note the order of inputs (different from odeint)!
def ydot(t, y, m, lambd, k):
    x, v = y[0], y[1]
    return array([v, -(lambd/m) * v - k * x / m])

m, lambd, k = 1.0, 0.1, 1.0   # Oscillator parameters
y0 = array([1.0, 5.0])        # Initial conditions [x, v]
t  = linspace(0.0, 50.0, 100) # Output times

## Set up the ODE object
r = ode(ydot)
r.set_integrator('dopri5')    # A Runge-Kutta solver
r.set_initial_value(y0)
r.set_f_params(m, lambd, k)

## Perform the integration.  Note that the "integrate" method only integrates
## up to one single final time point, rather than an array of times.
x    = zeros(len(t))
x[0] = y0[0]
for n in range(1,len(t)):
    r.integrate(t[n])
    assert r.successful()
    x[n] = (r.y)[0]

## Plot x versus t
```

```
plt.plot(t, x, 'b-')
plt.xlabel('t')
plt.ylabel('x')
plt.show()
```

See the documentation for a more detailed list of options, including the list of ODE solvers that you can choose from.

CHAPTER OVERVIEW

11: Discrete Fourier Transforms

The **Discrete Fourier Transform** (DFT) is a discretized version of the Fourier transform, which is widely used in numerical simulation and analysis. Given a set of N numbers $\{f_0, f_1, \ldots, f_{N-1}\}$, the DFT produces another set of N numbers N numbers $\{F_0, F_1, \ldots, F_{N-1}\}$, defined as follows:

$$\mathrm{DFT}\Big\{f_0, f_1, \ldots, f_{N-1}\Big\} = \Big\{F_0, F_1, \ldots, F_{N-1}\Big\} \qquad \text{where} \quad F_n = \sum_{m=0}^{N-1} e^{-2\pi i \frac{mn}{N}}\, f_m. \tag{11.1}$$

The inverse of this transformation is the Inverse Discrete Fourier Transform (IDFT):

$$\mathrm{IDFT}\Big\{F_0, F_1, \ldots, F_{N-1}\Big\} = \Big\{f_0, f_1, \ldots, f_{N-1}\Big\} \qquad \text{where} \quad f_m = \frac{1}{N}\sum_{n=0}^{N-1} e^{2\pi i \frac{mn}{N}}\, F_n. \tag{11.2}$$

The inverse relationship between the DFT and the IDFT is straightforward to prove, by using the identity

$$\sum_{m=0}^{N-1} e^{\pm 2\pi i \frac{m(n-n')}{N}} = N\delta_{nn'}, \tag{11.3}$$

where $\delta_{nn'}$ denotes the Kronecker delta. This identity is derived from the geometric series formula.

11.1: Conversion of Continuous Fourier Transform to DFT

The DFT is commonly encountered when discretizing formulas involving Fourier integrals. Recall the definition of the Fourier transform: given a function $f(x)$, where $x \in (-\infty, \infty)$, the Fourier transform is a function $F(k)$, where $k \in (-\infty, \infty)$, and these two functions are related by a pair of integral formulas:

$$F(k) = \int_{-\infty}^{\infty} dx\ e^{-ikx}\ f(x) \tag{11.1.1}$$

$$f(x) = \int_{-\infty}^{\infty} \frac{dk}{2\pi}\ e^{ikx}\ F(k). \tag{11.1.2}$$

Typically, a computer simulation or experimental measurement will produce values of $f(x)$ at certain values of x that are discrete and evenly-spaced. Suppose these points are $\{x_0, x_1, \ldots, x_{N-1}\}$, where the spacing is $\Delta x = x_{m+1} - x_m$; the corresponding data points are $\{f(x_0), \ldots, f(x_{N-1})\}$. We are then interested in finding the Fourier spectrum, i.e. plotting either $|F(k)|$ or $|F(k)|^2$ versus k. To do this, we can approximate the Fourier integral by using the mid-point rule:

$$F(k) \approx \Delta x \sum_{m=0}^{N-1} e^{-ikx_m}\ f(x_m). \tag{11.1.3}$$

Note that this necessitates a truncation of the Fourier integral. The Fourier integral ran over $-\infty < x < \infty$, but our numerical integral runs over a finite range $x_0 \lesssim x \lesssim x_{N-1}$. This truncation will have important consequences later. Now, we have to decide the values of k at which to find $F(k)$. Let us choose a set of N equally-spaced points,

$$k_n \equiv \frac{2\pi n}{N\Delta x}. \tag{11.1.4}$$

At these points, the discretized Fourier integral takes the form

$$F(k_n) \approx \Delta x \sum_{m=0}^{N-1} \exp\left[-\frac{2\pi i n(x_0 + m\Delta x)}{N\Delta x}\right] f(x_m) \tag{11.1.5}$$

$$= \Delta x\ \exp\left[-\frac{2\pi i n x_0}{N\Delta x}\right] \sum_{m=0}^{N-1} e^{-i2\pi nm/N}\ f(x_m) \tag{11.1.6}$$

$$= \Delta x\ \exp\left[-\frac{2\pi i n x_0}{N\Delta x}\right] \mathrm{DFT}\{f(x_m)\}_n. \tag{11.1.7}$$

Here $\mathrm{DFT}\{f(x_m)\}_n$ denotes the n-th element of the Discrete Fourier Transform (DFT). The m index inside the curly brackets is a dummy index, indicating that the DFT involves an internal sum over this index (we're slightly abusing mathematical notation here). The phase factor, $\exp[-2\pi i n x_0/N\Delta x]$, is determined by the choice of "origin" for the spatial coordinates; it does not affect $|F(k_n)|^2$ (which is what's used to plot the Fourier spectrum).

The DFT and IDFT can be computed very efficiently, in $O(N \log N)$ time, using an algorithm called the Fast Fourier Transform (FFT). We will not discuss the FFT algorithm in this article, but many good explanations can be found elsewhere online. In Python, you can perform an FFT (fast DFT) by calling `fft` , and an inverse FFT (fast IDFT) by calling `ifft`

11.2: Spectral Resolution and Range

In the previous section, we showed how a continuous Fourier integral is converted into a DFT. This process involved two distinct approximations. Firstly, the Fourier integral is *truncated* from its original range, $x \in (-\infty, \infty)$, to a finite interval of length $N\Delta x$. Secondly, the integral is *discretized* by reducing the continuous variable x to a set of discrete points $\{x_0, \ldots, x_{N-1}\}$. Both of these approximations have important consequences for the accuracy of our numerical Fourier spectrum, which we will examine in turn.

11.2.1 Spectral Resolution

The truncation of the Fourier integral limits the **spectral resolution** of the Fourier spectrum. To see this, suppose we perform truncation without discretization, by taking a continuous Fourier integral and truncating it to a finite range $x \in [0, X]$:

$$F(k) \approx \int_0^X dx\, e^{-ikx}\, f(x). \tag{11.2.1}$$

Consider a harmonic function $f(x) = e^{ik_0 x}$. The exact Fourier transform can be shown to be a delta function, $F(k) = 2\pi\,\delta(k - k_0)$, i.e. an infinitely sharp peak centered at $k = k_0$. With the above truncation, however, the resulting integral is

$$F(k) \approx \int_0^X dx\, e^{-i(k-k_0)x} = \frac{2\sin[(k-k_0)X/2]}{k-k_0} \cdot e^{-i(k-k_0)X/2}. \tag{11.2.2}$$

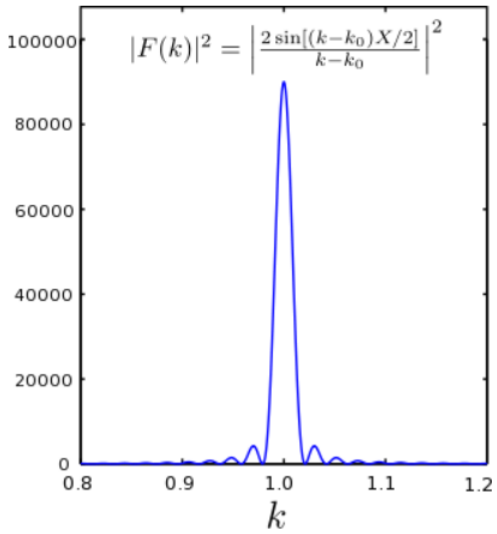

Figure 11.2.1: Fourier power spectrum from a truncated Fourier transform of $f(x) = \exp(ik_0 x)$, with $k_0 = 1$ and sampling interval $k \in [0, 300]$ (about 48 periods).

For $X \to \infty$, the above formula approaches a delta function (an infinitesimally-thin peak) centered at $k = k_0$. But for finite X, the plot of $|F(k)|^2$ versus ω behaves as shown in Fig 11.2.1. Evidently, truncating the Fourier integral has "smeared out" the Fourier spectrum, broadening the infinitesimally-thin delta function peak into a finite-width peak. The peak width, $\Delta k \sim 1/X$, limits the "resolution" of our Fourier analysis.

In the discretized Fourier transform, the truncation of the Fourier integral has essentially the same effect. As discussed in the previous section, the DFT is defined at $k_n \equiv 2\pi n/X$; hence, the resolution of the Fourier spectrum is $\Delta k = 2\pi/X$.

11.2.2 Spectral Range

The other approximation which we made in going from a continuous Fourier transform to the DFT involved sampling $f(x)$ at discrete values of x. This discretization has the effect of limiting the **spectral range**. To see this, let us look again at the DFT formula, which is dimensionless:

$$F_n = \sum_{m=0}^{N-1} e^{-2\pi i \frac{mn}{N}}\, f_m. \tag{11.2.3}$$

Normally, we consider only the indices $n = 0, 1, \ldots, N-1$. However, if we replace n with $n+N$ in the right-hand side, the result would be the same:

$$F_{n+N} = \sum_{m=0}^{N-1} e^{-2\pi i \frac{m(n+N)}{N}} f_m = \sum_{m=0}^{N-1} e^{-2\pi i \frac{mn)}{N} - 2\pi i m} f_m = F_n. \tag{11.2.4}$$

We can hence regard F as a periodic discrete function of n, with period N. Next, consider how the DFT is related to the physical x and k variables. Taking $x_0 = 0$ for simplicity,

$$F(k_n) = \sum_{m=0}^{N-1} e^{-2\pi i \frac{mn}{N}} f(x_m) = \sum_{m=0}^{N-1} e^{-ik_n \cdot m\Delta x} f(x_m). \tag{11.2.5}$$

If we perform the replacement

$$k_n \to k_n + K, \quad \text{where } K \equiv \frac{2\pi}{\Delta x}, \tag{11.2.6}$$

then evidently $F(k_n)$ is left unchanged. Indeed, we could add any integer multiple of K without altering the result. This means that the DFT spectrum is only defined under k modulo K, by contrast with the continuous Fourier transform which is defined over the entire interval $-\infty < k < \infty$.

The default definition of the DFT gives the integer indices $n = 0, 1, \ldots, N-1$, which corresponds to $0 \leq k \lesssim K$. However, when plotting the DFT spectrum, we usually adjust the range of k to $-K/2 \lesssim k \lesssim K/2$. This is done by taking the "upper half" of the DFT spectrum, $K/2 \lesssim k \lesssim K$, and translating it via the replacement $k \to k - K$. Due to the periodicity of the DFT, the upper half of the DFT spectrum becomes the negative k part of the spectrum. In terms of the integer indices n, the process is depicted in the figure below:

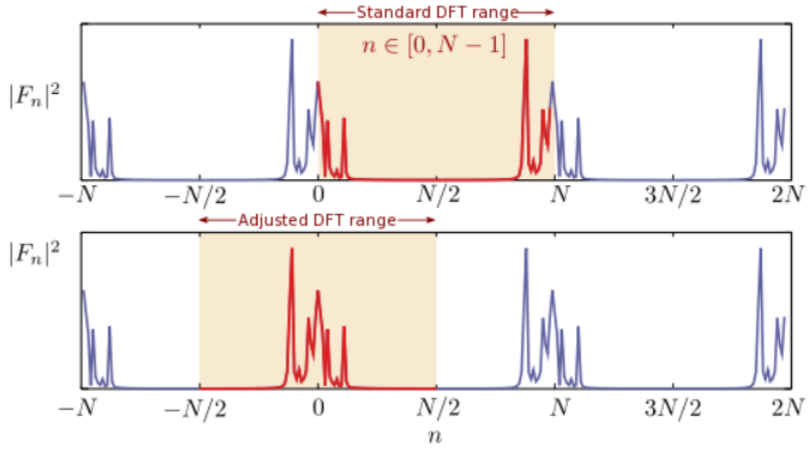

Figure 11.2.2: A DFT spectrum, F_n, is periodic with period N. By default, the DFT is reported in the spectral range $n \in [0, N-1]$ (red curve in the upper plot). To relate this to the continuous Fourier transform, we re-center the spectrum at $n = 0$, which is equivalent to translating the upper half of the spectrum to negative values (red curve in the lower plot).

The reason for this adjustment is that, intuitively, the discretized Fourier spectrum contains information about the "low-frequency" part of the spectrum, $|k| < K/2$, including both positive and negative values of k. On the other hand, the discretized Fourier spectrum lacks information about the "high frequency" part of the spectrum, which correspond to harmonics with periods shorter than the discretization step Δx. Hence, it makes sense to "center" our Fourier spectrum around the origin. It can then be shown that as the discretization step approaches zero (and hence $K = 2\pi/\Delta x \to \infty$), the $|k| \ll K$ part of the adjusted DFT spectrum converges to the exact (continuous) Fourier spectrum.

The corollary to the above discussion is that if we have a function which has no frequency components larger than $k_{\max}$, then it is sufficient to use a sampling interval $\Delta x = \pi/k_{\max}$. This is called the Nyquist-Shannon sampling theorem.

11.2.3 Summary of Spectral Relations

The results of the previous sections can be summarized in this way:

- The total range of x, which is denoted by X, limits the resolution of the spectrum to $\Delta k = 2\pi/X$.

- The resolution of x, which is the discretization step Δx, limits the range of the spectrum to $K = 2\pi/\Delta x$.

These relations are easy to remember, because the "interval length" in the one domain places a limit on the "discretization step" in the other domain. It is very important to keep these relations in mind when working with discrete Fourier transforms! For example, a common mistake that people make is to try to improve the resolution of a Fourier spectrum by increasing the number of discretization steps, N, while keeping the total interval X fixed. This doesn't work; it leaves the spectral resolution unchanged! In order to improve the spectral resolution, one has to increase the total interval instead.

11.3: The Split-Step Fourier Method

As an example of the usefulness of the DFT, let us discuss a DFT-based method for performing numerical integration of a *partial* differential equation, known as the **split-step Fourier method**. Here, the method will be presented in the context of the time-dependent Schrödinger equation in 1D space:

$$i\frac{d\psi(x,t)}{dt} = \left[-\frac{1}{2}\frac{\partial^2}{\partial x^2} + V(x,t)\right]\psi(x,t) \tag{11.3.1}$$

We have taken $\hbar = m = 1$ for simplicity. At each time, the wavefunction is a continuous function of x. Let us truncate and discretize this spatial coordinate, by defining a computational domain of length L containing N discretization points:

$$\psi_n(t) = \psi(x_n,t), \quad \text{where} \quad x_n = -\frac{L}{2} + n\Delta x, \quad \Delta x = \frac{L}{N}. \tag{11.3.2}$$

Thus, the wavefunction at each time is represented by a complex vector, which we call a "state vector":

$$\vec{\psi}(t) = \begin{pmatrix} \psi_0(t) \\ \psi_1(t) \\ \vdots \\ \psi_{N-1}(t) \end{pmatrix}. \tag{11.3.3}$$

Given an initial state vector $\vec{\psi}(t_a)$, the problem is to compute $\vec{\psi}(t_b)$ at a later time t_b. Note that this differs from previously-studied numerical ODE problems in one important respect: evolving in time involves taking second-order *spatial* derivatives. We won't go into the details, but it turns out that standard methods for time-stepping and discretizing space don't work very well here, because the errors from time-stepping and spatial discretization interact badly with one another. The split-step Fourier method provides a better way to solve the problem.

11.3.1 Factorizing the Time-Evolution Operator

The split-step Fourier method is based on the concept of the *time-evolution operator*. Given a wavefunction $\psi(x,t_j)$, the wavefunction after a small time step τ is

$$\psi(x,t_j+\tau) = U(t_j+\tau|t_j)\,\psi(x,t_j), \quad \text{where} \quad U(t_j+\tau|t_j) \approx \exp\left\{-i\tau\left[-\frac{1}{2}\frac{\partial^2}{\partial x^2} + V(x,t_j+\tau/2)\right]\right\}. \tag{11.3.4}$$

Here, the $\exp(\cdots)$ refers to the exponential of an operator (one involving spatial derivatives). We call $U(t_b|t_a)$ the time-evolution operator, which evolves the system from time t_a to t_b. The exponential of any operator $\mathcal{A}$ is defined as the infinite series

$$\exp(\mathcal{A}) = I + \mathcal{A} + \frac{1}{2}\mathcal{A}^2 + \frac{1}{6}\mathcal{A}^3 + \cdots \tag{11.3.5}$$

In this case, the exponential contains the Hamiltonian, which consists of a kinetic energy term and a potential energy term that do not generally commute. Due to this non-commutivity, the exponential cannot be simplified by factorization:

$$\exp\left\{-i\tau\left[-\frac{1}{2}\frac{\partial^2}{\partial x^2} + V(x,t_j+\tau/2)\right]\right\} \;\neq\; \exp\left\{\frac{i\tau}{2}\left[\frac{\partial^2}{\partial x^2}\right]\right\}\exp\left\{-i\tau V(x,t_j+\tau/2)\right\}. \tag{11.3.6}$$

However, we can obtain an approximate factorization by making use of the series definition of the exponential of an operator. One can show that

$$\exp(\mathcal{A}+\mathcal{B}) = \exp(\mathcal{A}/2)\exp(\mathcal{B})\exp(\mathcal{A}/2) + O\Big((\mathcal{A},\mathcal{B})^3\Big), \tag{11.3.7}$$

which is a variant of an important formula known as the Baker–Campbell–Hausdorff formula. On the right-hand side, note that $\exp(\mathcal{B})$ is sandwiched "symmetrically" between two copies of $\exp(\mathcal{A}/2)$. This symmetric arrangement reduces the approximation error to third order, by the cancellation of lower-order errors (in a manner similar to the mid-point formula for the discretized derivative). Applying this factorization to the time-evolution operator gives

$$\boxed{U(t_j+\tau|t_j) \;\approx\; U_{\mathcal{K}} \cdot U_{\mathcal{V}}(t_j+\tau/2) \cdot U_{\mathcal{K}}, \quad \text{where} \quad \begin{cases} U_{\mathcal{K}} & = \exp\left[\frac{i\tau}{4}\frac{d^2}{dx^2}\right] \\ U_{\mathcal{V}}(t) & = \exp\left[-i\tau\, V(x,\,t)\right] \end{cases}} \tag{11.3.8}$$

In other words, the time-evolution operator decomposes into three pieces. That's why we call this a "split-step" algorithm: each time step from t_j to $t_j+\tau$ consists of applying a kinetic step, then applying a potential step, then applying another kinetic step, in sequence. As previously noted, we'll be working with state vectors (complex N-component vectors), defined through spatial discretization of the wavefunction. So we need to figure out how the above stepping operators act on these state vectors:

$$U_{\mathcal{K}}\,\vec{\psi} \;=\; ??, \qquad U_{\mathcal{V}}\,\vec{\psi} \;=\; ?? \tag{11.3.9}$$

The potential stepping operator is simple to deal with. Since the state vector represents the wavefunction at different points in space, the potential operator is represented by a diagonal matrix, and its exponential is also diagonal:

$$U_{\mathcal{V}} \begin{pmatrix} \psi_0 \\ \vdots \\ \psi_{N-1} \end{pmatrix} \;=\; \begin{pmatrix} \exp\left[-i\tau V(x_0, t+\frac{\tau}{2})\right] & & \\ & \ddots & \\ & & \exp\left[-i\tau V(x_{N-1}, t+\frac{\tau}{2})\right] \end{pmatrix} \begin{pmatrix} \psi_0 \\ \vdots \\ \psi_{N-1} \end{pmatrix} \tag{11.3.10}$$

11.3.2 Kinetic Step

The kinetic stepping operator, $U_{\mathcal{K}}$, is less obvious. It contains spatial derivatives and is thus *not* diagonal in the current basis. The key thing to realize, however, is that this operator is diagonal in *wavenumber space*. Let us return to the continuous wavefunction, and write its Fourier representation:

$$\psi(x) = \int_{-\infty}^{\infty} \frac{dk}{2\pi}\; e^{ikx}\; \Psi_k. \tag{11.3.11}$$

Then

$$U_{\mathcal{K}}\;\psi(x) = \int_{-\infty}^{\infty} \frac{dk}{2\pi}\; \exp\left(-\frac{i\tau}{4}k^2\right)\; e^{ikx}\; \Psi_k. \tag{11.3.12}$$

Let us discretize space in steps of Δx, as discussed earlier, and also discretize the Fourier integrals by steps of Δk:

$$x_n = -\frac{L}{2} + n\Delta x, \tag{11.3.13}$$

$$k_n = -\frac{K}{2} + n\Delta k \tag{11.3.14}$$

The values of K and Δk will be chosen shortly. The discretized integrals become

$$\psi_m \approx \sum_{n=0}^{N-1} \frac{\Delta k}{2\pi}\; e^{ik_n x_m}\; \Psi_{k_n}, \tag{11.3.15}$$

$$(U_{\mathcal{K}}\psi)_m \approx \sum_{n=0}^{N-1} \frac{\Delta k}{2\pi}\; e^{-\frac{i\tau}{4}k_n^2}\; e^{ik_n x_m}\; \Psi_{k_n}. \tag{11.3.16}$$

Let us now choose the k-space discretization parameters to be

$$\Delta k = \frac{2\pi}{N\Delta x} = \frac{2\pi}{L}, \quad K = N\Delta k = \frac{2\pi}{\Delta x} \quad \Rightarrow \quad k_n = -\frac{\pi}{\Delta x} + \frac{2\pi n}{N\Delta x}. \tag{11.3.17}$$

With this choice, we can show with a bit of algebra that the integral for ψ_m reduces to an IDFT:

$$\psi_m = (-1)^m \frac{1}{N} \sum_{n=0}^{N-1} \left(\frac{1}{\Delta x} e^{iN\pi/2}\; e^{-in\pi}\; \Psi_{k_n}\right) e^{\frac{2\pi i m n}{N}} \tag{11.3.18}$$

$$= (-1)^m\; \text{IDFT}\left\{\frac{1}{\Delta x} e^{iN\pi/2}\; (-1)^n\; \Psi_{k_n}\right\}_m \tag{11.3.19}$$

$$\Rightarrow \Psi_{k_n} = \Delta x\, e^{-iN\pi/2}\; (-1)^n\;\; \text{DFT}\left\{(-1)^m\; \psi_m\right\}_n \tag{11.3.20}$$

Likewise,

$$(U_{\mathcal{K}}\ \psi)_m = (-1)^m\ \mathrm{IDFT}\left\{ \frac{1}{\Delta x} e^{iN\pi/2}\ (-1)^n e^{-\frac{i\tau}{4}k_n^2}\ \Psi_{k_n} \right\}_m \tag{11.3.21}$$

Putting these results together, we get

$$(U_{\mathcal{K}}\ \psi)_m = (-1)^m\ \mathrm{IDFT}\left\{ e^{-\frac{i\tau}{4}k_n^2}\ \mathrm{DFT}\Big\{ (-1)^p\, \psi_p \Big\}_n \right\}_m \tag{11.3.22}$$

$$\text{where}\ \ k_n = -\frac{\pi N}{L} + \frac{2\pi n}{L} \tag{11.3.23}$$

Hence, the $U_{\mathcal{K}}$ kinetic stepping operator can be implemented by taking a DFT, multiplying the resulting vector elements by $\exp(-i\tau k_n^2/4)$ phase factors, and taking an IDFT. The runtime of the stepping process is $O(N\log(N))$. The m, n, and p indices all run over the range $[0, 1, \cdots, N-1]$, consistent with the standard definition of the DFT and IDFT.

CHAPTER OVERVIEW

12: Markov Chains

A **Markov chain** refers to a sequence (or "chain") of discrete events, generated according to a fixed set of probabilistic rules. The most important property of these rules is that they can only refer to the current state of the system, and cannot depend on the past states of the system. Markov chains have numerous applications in physics, mathematics, and computing. In statistical mechanics, for instance, Markov chains are used to describe the random sequence of micro-states visited by a system undergoing thermal fluctuations.

12.1: The Simplest Markov Chain- The Coin-Flipping Game

12.1.1 Game Description

Before giving the general description of a Markov chain, let us study a few specific examples of simple Markov chains. One of the simplest is a "coin-flip" game. Suppose we have a coin which can be in one of two "states": heads (H) or tails (T). At each step, we flip the coin, producing a new state which is H or T with equal probability. In this way, we generate a sequence like "HTTHTHTTHH..." If we run the game again, we would generate another different sequence, like "HTTTTHHTTH..." Each of these sequences is a Markov chain.

This process can be visualized using a "state diagram":

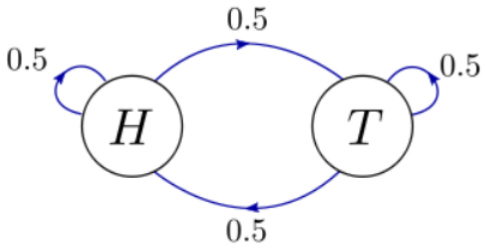

Figure 12.1.1: State diagram for a fair coin-flipping game.

Here, the two circles represent the two possible states of the system, "H" and "T", at any step in the coin-flip game. The arrows indicate the possible states that the system could transition into, during the next step of the game. Attached to each arrow is a number giving the probability of that transition. For example, if we are in the state "H", there are two possible states we could transition into during the next step: "T" (with probability 0.5), or "H" (with probability 0.5). By conservation of probability, the transition probabilities coming out of each state must sum up to one.

Next, suppose the coin-flipping game is unfair. The coin might be heavier on one side, so that it is overall more likely to land on H than T. It might also be slightly more likely to land on the same face that it was flipped from (real coins actually do behave this way). The resulting state diagram can look like this:

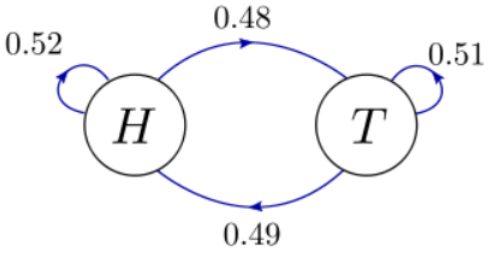

Figure 12.1.2: State diagram for a (particular) unfair coin-flipping game.

Notice that the individual transition probabilities are no longer 0.5, reflecting the aforementioned unfair effects. However, the transition probabilities coming out of each state still sum to 1 ($0.52+0.48$ coming out of "H", and $0.51+0.49$ coming out of "T").

12.1.2 State Probabilities

If we play the above unfair game many times, H and T will tend to occur with slightly different probabilities, not exactly equal to 0.5. At a given step, let p_H denote the probability to be in state H, and p_T the probability to be in state T. Let $P(T|H)$ denote the transition probability for going from H to T, during the next step; and similarly for the other three possible transitions. According to Bayes' rule, we can write the probability to get H on the next step as

$$p'_H = P(H|H)p_H + P(H|T)p_T. \tag{12.1.1}$$

Similarly, the probability to get T on the next step is

$$p'_T = P(T|H)p_H + P(T|T)p_T. \tag{12.1.2}$$

We can combine these into a single matrix equation:

$$\begin{bmatrix} p'_H \\ p'_T \end{bmatrix} = \begin{bmatrix} P(H|H) & P(H|T) \\ P(T|H) & P(T|T) \end{bmatrix} \begin{bmatrix} p_H \\ p_T \end{bmatrix} = \begin{bmatrix} 0.52 & 0.49 \\ 0.48 & 0.51 \end{bmatrix} \begin{bmatrix} p_H \\ p_T \end{bmatrix}. \tag{12.1.3}$$

The matrix of transition probabilities is called the *transition matrix*. At the beginning of the game, we can specify the coin state to be (say) H, so that $p_H = 1$ and $p_T = 0$. If we multiply the vector of state probabilities by the transition matrix, that gives the state probabilities for the next step. Multiplying by the transition matrix K times gives the state probabilities after K steps.

After a large number of steps, the states probabilities might converge to a "stationary" distribution, such that they no longer change significantly on subsequent steps. Let these stationary probabilities be denoted by $\{\pi_H, \pi_T\}$. According to the above equation for transition probabilities, the stationary probabilities must satisfy

$$\begin{bmatrix} \pi_H \\ \pi_T \end{bmatrix} = \begin{bmatrix} P(H|H) & P(H|T) \\ P(T|H) & P(T|T) \end{bmatrix} \begin{bmatrix} \pi_H \\ \pi_T \end{bmatrix}. \tag{12.1.4}$$

This system of linear equations can be solved by brute force (we'll discuss a more systematic approach later). The result is

$$\pi_H = \frac{P(H|T)}{P(T|H)+P(H|T)}, \quad \pi_T = \frac{P(T|H)}{P(T|H)+P(H|T)} \qquad \text{(stationary distribution)}. \tag{12.1.5}$$

Plugging in the numerical values for the transition probabilities, we end up with $\pi_H = 0.50515,\ \pi_T = 0.49485$

12.2: General Description

12.2.1 Markov Processes

More generally, suppose we have a system possessing a discrete set of states, which can be labeled by an integer $0, 1, 2, \ldots$ A **Markov process** is a set of probabilistic rules that tell us how to choose a new state of the system, based on the system's current state. If the system is currently in state n, then the probability of choosing state m on the next step is denoted by $P(m|n)$. We call this the "transition probability" from state n to state m. By repeatedly applying the Markov process, we move the system through a random sequence of states, $\{n^{(0)}, n^{(1)}, n^{(2)}, n^{(3)}, \ldots\}$, where $n^{(k)}$ denotes the state on step k. This kind of random sequence is called a **Markov chain**.

There is an important constraint on the transition probabilities of the Markov process. Because the system must transition to *some* state on each step,

$$\sum_m P(m|n) = 1 \quad \text{for all } n \in \{0, 1, \ldots\}. \tag{12.2.1}$$

Next, we introduce the idea of **state probabilities**. Suppose we look at the ensemble of all possible Markov chains which can be generated by a given Markov process. Let $\{p_0^{(k)}, p_1^{(k)}, p_2^{(k)}, \ldots\}$ denote the probabilities for the various states, $n = 0, 1, 2, \ldots$, on step k. Given these, what are the probabilities for the various states on step $k+1$? According to Bayes' theorem, we can write $p_m^{(k+1)}$ as a sum over conditional probabilities:

$$p_m^{(k+1)} = \sum_n P(m|n)\, p_n^{(k)}. \tag{12.2.2}$$

This has the form of a matrix equation:

$$\begin{bmatrix} p_0^{(k+1)} \\ p_1^{(k+1)} \\ \vdots \end{bmatrix} = \begin{bmatrix} P(0|0) & P(0|1) & \cdots \\ P(1|0) & P(1|1) & \cdots \\ \vdots & \vdots & \end{bmatrix} \begin{bmatrix} p_0^{(k)} \\ p_1^{(k)} \\ \vdots \end{bmatrix}, \tag{12.2.3}$$

where the matrix on the right-hand side is called the **transition matrix**. Each element of this matrix is a real number between 0 and 1; furthermore, because of the aforementioned conservation of transition probabilities, each column of the matrix sums to 1. In mathematics, matrices of this type are called "left stochastic matrices".

12.2.2 Stationary Distribution

A **stationary distribution** is a set of state probabilities $\{\pi_0, \pi_1, \pi_2, \ldots\}$, such that passing through one step of the Markov process leaves the probabilities unchanged:

$$\pi_m = \sum_n P(m|n)\, \pi_n. \tag{12.2.4}$$

By looking at the equivalent matrix equation, we see the vector $[\pi_0; \pi_1; \pi_2; \ldots]$ must be an eigenvector of the transition matrix, with eigenvalue 1. It turns out that there is a mathematical theorem (the Perron–Frobenius theorem) which states every left stochastic matrix has an eigenvector of this sort. Hence, every Markov process possesses a stationary distribution. Stationary distributions are the main reasons we are interested in Markov processes. In physics, we are often interested in using Markov processes to model thermodynamic systems, such that a stationary distribution represents the distribution of thermodynamic micro-states under thermal equilibrium. (We'll see an example in the next section.) Knowing the stationary distribution, we can figure out all the thermodynamic properties of the system, such as its average energy.

In principle, one way to figure out the stationary distribution is to construct the transition matrix, solve the eigenvalue problem, and pick out the eigenvector with eigenvalue 1. The trouble is that we are often interested in systems where the number of possible states is *huge*—in some cases, larger than the number of atoms in the universe! In such cases, it is not possible to explicitly generate the transition matrix, let alone solve the eigenvalue problem.

We now come upon a happy and important fact: for a huge class of Markov processes, the distribution of states within a sufficiently long Markov chain will converge to the stationary distribution. Hence, in order to find out about the stationary distribution, we

simply need to generate a long Markov chain, and study its statistical properties.

12.3: The Ehrenfest Model

12.3.1 Model Description

The coin-flipping game is a "two-state" Markov chain. For physics applications, we're often interested in Markov chains where the number of possible states is huge (e.g. thermodynamic microstates). The *Ehrenfest model* is a nice and simple example which illustrates many of the properties of such Markov chains. This model was introduced by the husband-and-wife physicist team of Paul and Tatyana Ehrenfest in 1907, in order to study the physics of diffusion.

Suppose we have two boxes, labeled A and B, and a total of N distinguishable particles to distribute between the two boxes. At a given point in time, let there be n particles in box A, and hence $N-n$ particles in box B. Now, we repeatedly apply the following procedure:

1. Randomly choose one of the N particles (with equal probability).
2. With probability q, move the chosen particle from whichever box it happens to be into the other box. Otherwise (with probability $1-q$), leave the particle in its current box.

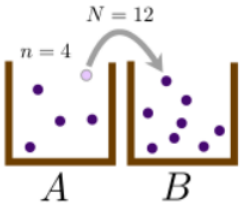

Figure 12.3.1: Schematic of the Ehrenfest model.

If there are n particles in box A, then we have probability n/N of choosing a particle in box A, followed by a probability of q to move that particle into box B. Following similar logic for all the other possibilities, we arrive at three possible outcomes:

- Move a particle from A to B: probability nq/N
- Move a particle from B to A: probability $(N-n)q/N$
- Leave the system unchanged: probability $1-q$

You can check that (i) the probabilities sum up to 1, and (ii) this summary holds true for the end-cases $n=N$ and $n=0$.

12.3.2 Markov Chain Description

We can label the states of the system using an integer $n \in \{0, 1, \ldots, N\}$, corresponding to the number of particles in box A. There are $N+1$ possible states, and the state diagram is as follows:

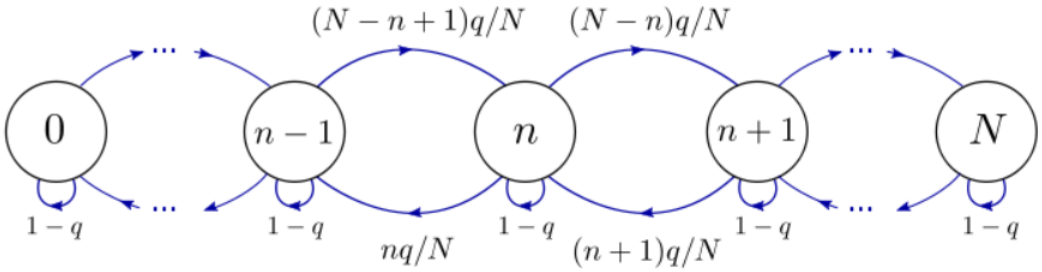

Figure 12.3.2: State diagram for the Ehrenfest model.

Suppose we start out in state $n_0 = N$, by putting all the particles in box A. As we repeatedly apply the Ehrenfest procedure, the system goes through a sequence of states, $\{n_0 = N, n_1, n_2, n_3, \ldots\}$, which can be described as a Markov chain. Plotting the state n_k versus the step number k, we see a random trajectory like the one below:

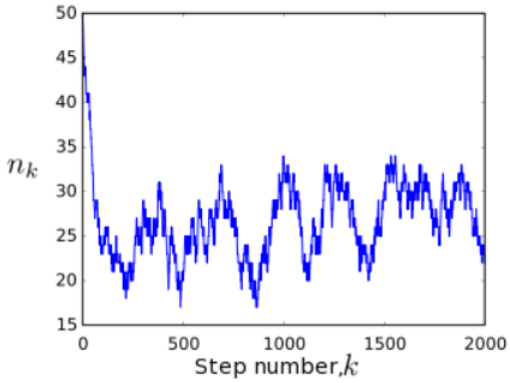

Figure 12.3.3: Sample trajectory of an Ehrenfest model with $N = 50$ and $q = 0.5$.

Notice that the system moves rapidly away from its initial state, $n = 50$, and settles into a behavior where it fluctuates around the mid-point state $n = 25$. Let us look for the stationary distribution, in which the probability of being in each state is unchanged on subsequent steps. Let π_n denote the stationary probability for being in state n. According to Bayes' rule, this probability distribution needs to satisfy

$$\pi_n = P(n|n-1)\pi_{n-1} + P(n|n)\pi_n + P(n|n+1)\pi_{n+1} \tag{12.3.1}$$

$$= \frac{N-n+1}{N}\, q\, \pi_{n-1} + (1-q)\pi_n + \frac{n+1}{N}\, q\, \pi_{n+1}. \tag{12.3.2}$$

We can figure out π_n using two different methods. The first method is to use our knowledge of statistical mechanics. In the stationary distribution, each individual particle should have an equal chance of being in box A or box B. There are 2^N possible box assignments, each of which is energetically equivalent and hence have equal probabilities. Hence, the probability of finding n particles in box A is the number of ways of picking n particles, which is $\binom{N}{n}$, divided by the number of possible box assignments. This gives

$$\pi_n = \binom{N}{n} 2^{-N}. \tag{12.3.3}$$

Substituting into the Bayes' rule formula, we can verify that this distribution is indeed stationary. Note that π_n turns out to be independent of q (the probability of transferring a chosen particle to the other box). Intuitively, q governs how "quickly" we are transferring particles from one box to the other. Therefore, it should affect how quickly the system reaches its stationary or "equilibrium" behavior, but not the stationary distribution itself.

12.3.3 Detailed Balance

There is another way to figure out π_n, which doesn't rely on guessing the answer in one shot. Suppose we pick a pair of neighboring states, n and $n+1$, and assume that the rate at which the $n \to n+1$ transition occurs is the same as the rate at which the opposite transition, $n \to n+1$, occurs. Such a condition is not guaranteed to hold, but if it holds *for every pair of states*, then the probability distribution is necessarily stationary. This situation is called *detailed balance*. In terms of the state probabilities and transition probabilities, detailed balance requires

$$P(n+1|n)\,\pi_n = P(n|n+1)\,\pi_{n+1} \qquad \forall n \in \{0, \ldots, N\}, \tag{12.3.4}$$

for this Markov chain. Plugging in the transition probabilities, we obtain the recursion relation

$$\pi_{n+1} = \frac{N-n}{n+1}\,\pi_n. \tag{12.3.5}$$

What's convenient about this recursion relation is that it only involves π_n and π_{n+1}, unlike the Bayes' rule relation which also included π_{n-1}. By induction, we can now easily show that

$$\pi_n = \binom{N}{n}\pi_0. \tag{12.3.6}$$

By conservation of probability, $\sum_n \pi_n = 1$, we can show that $\pi_0 = 2^{-N}$. This leads to

$$\pi_n = \binom{N}{n} 2^{-N}, \tag{12.3.7}$$

which is the result that we'd previously guessed using purely statistical arguments.

For more complicated Markov chains, it may not be possible to guess the stationary distribution; in such cases, the detailed balance argument is often the best approach. Note, however, that the detailed balance condition is not guaranteed to occur. There are some Markov chains which do not obey detailed balance, so we always need to verify that the detailed balance condition's result is self-consistent (i.e., that it can indeed be obeyed for every pair of states).

CHAPTER OVERVIEW

13: The Markov Chain Monte Carlo Method

The **Markov Chain Monte Carlo** (MCMC) method is a powerful computational technique based on Markov chains, which has numerous applications in physics as well as computer science.

13.1: Basic Formulation

The basic idea behind the MCMC method is simple. Suppose we have a set of states labeled by an integer index $n \in \{0, 1, 2, \ldots\}$, where each state is associated with a probability π_n. For example, in statistical mechanics, for a system maintained at a constant temperature T, each state occurs with probability

$$\pi_n = \frac{\exp\left(-\frac{E_n}{kT}\right)}{Z}, \tag{13.1.1}$$

where E_n is the energy of the state, k is Boltzmann's constant, and Z is the partition function

$$Z = \sum_n \exp\left(-\frac{E_n}{kT}\right). \tag{13.1.2}$$

From $\{\pi_n\}$, we would like to calculate various expectation values, which describe the thermodynamic properties of the system. For example, we might be interested in the average energy, which is defined as

$$\langle E \rangle = \sum_n E_n \pi_n. \tag{13.1.3}$$

The most straightforward way to find $\langle E \rangle$ is to explicitly calculate the above sum. But if the number of states is very large, this is prohibitively time-consuming (unless there is a tractable analytic solution, and frequently there isn't). For example, if we are interested in describing 1000 distinct atoms each having 2 possible energy levels, the total number of states is $2^{1000} \approx 10^{301}$. Trying to calculate a sum over this mind-boggingly many terms would take longer than the age of the universe.

The MCMC method gets around this problem by *selectively* sampling the states. To accomplish this, we *design* a Markov process whose stationary distribution is identically equal to the given probabilities π_n. We will discuss how to design the Markov process in the next section. Once we have an appropriate Markov process, we can use it to generate a long Markov chain, and use that chain to calculate moving averages of our desired quantities, like $\langle E \rangle$. If the Markov chain is sufficiently long, the average calculated this way will converge to the true expectation value $\langle E \rangle$.

The key fact which makes all this work is that the required length of the Markov chain is usually *much* less than the total number of possible states. For the above-mentioned problem of 1000 distinct two-level atoms, there are 10^{301} states, but a Markov chain of as few as 10^6 steps can get within several percent of the true value of $\langle E \rangle$. (The actual accuracy will vary from system to system.) The reason for this is that the vast majority of states are extremely unlikely, and their contributions to the sum leading to $\langle E \rangle$ are very small. A Markov chain can get a good estimate for $\langle E \rangle$ by sampling the states that have the highest probabilities, without spending much time on low-probability states.

13.1.1 The Metropolis Algorithm

The MCMC method requires us to design a Markov process to match a given stationary distribution $\{\pi_n\}$. This is an open-ended problem, and generally there are many good ways to accomplish this goal. The most common method, called the **Metropolis algorithm**, is based on the principle of detailed balance, which we discussed in the article on Markov chains. To recap, the principle of detailed balance states that under generic circumstances (which are frequently met in physics), a Markov process's transition probabilities are related to the stationary distribution by

$$P(n|m)\,\pi_m = P(m|n)\,\pi_n \quad \text{for all } m, n. \tag{13.1.4}$$

The Metropolis algorithm specifies the following Markov process:

1. Suppose that on step k, the system is in state n. Randomly choose a candidate state, m, by making an unbiased random step through the space of possible states. (Just how this choice is made is system-dependent, and we'll discuss this below.)
2. Compare the probabilities π_n and π_m:
   - If $\pi_m \geq \pi_n$, *accept* the candidate.
   - If $\pi_m < \pi_n$, *accept* the candidate with probability π_m / π_n. Otherwise, *reject* the candidate.
3. If the candidate is accepted, the state on step $k+1$ is m. Otherwise, the state on step $k+1$ remains n.
4. Repeat.

Based on the above description, let us verify that the stationary distribution of the Markov process satisfies the desired detailed-balance condition. Consider any two states a, b, and assume without loss of generality that $\pi_a \leq \pi_b$. If we start from a, suppose we choose a candidate step $a \to b$ with some probability q. Then, according to the Metropolis rules, the probability of actually making this transition, $a \to b$, is q times the acceptance probability 1. On the other hand, suppose we start from b instead. Because the candidate choice is unbiased, we will choose a candidate step $b \to a$ with the same probability q as in the previous case. Hence, the transition probability for $b \to a$ is q times the acceptance probability of π_a / π_b. As a result,

$$\begin{cases} P(b|a) = q \times 1 \\ P(a|b) = q \times \dfrac{\pi_a}{\pi_b}. \end{cases} \quad \Rightarrow \quad P(a|b)\,\pi_b = P(b|a)\,\pi_a \tag{13.1.5}$$

Since this reasoning holds for arbitrary a, b, the principle of detailed balance implies that the stationary distribution of our Markov process follows the desired distribution $\{\pi_n\}$.

Expression in Terms of Energies

In physics, the MCMC method is commonly applied to thermodynamic states, for which

$$\pi_n = \frac{\exp\left(-\frac{E_n}{kT}\right)}{Z}. \tag{13.1.6}$$

In such cases, the Metropolis algorithm can be equivalently expressed in terms of the state energies:

1. Suppose that on step k, the system is in state n. Randomly choose a candidate state, m, by making an unbiased random step through the space of possible states.
2. Compare the energies E_n and E_m:
   - If $E_m \leq E_n$, *accept* the candidate.
   - If $E_m > E_n$, *accept* the candidate with probability $\exp[(E_n - E_m)/kT]$. Otherwise, *reject* the candidate.
3. If the candidate is accepted, the state on step $k+1$ is m. Otherwise, the state on step $k+1$ remains n.
4. Repeat.

13.1.2 Stepping Through State Space

One way of thinking about the Metropolis algorithm is that it takes a scheme for performing an *unbiased* random walk through the space of possible states (represented by our candidate choices), and converts it into a scheme for performing a *biased* random walk. The biased random walk corresponds to a Markov process with the stationary distribution we are interested in.

The way the Metropolis candidates are chosen (i.e., the "unbiased random walk" part) varies from system to system, and once again there are multiple valid schemes that we could employ. For example, suppose we have a collection of 6 atoms, where each atom can be in the level labeled 0 or the level labeled 1. Each state of the overall system is described by a list of 6 symbols, e.g. 011001. Then we can make an unbiased walk through the "state space" by randomly choosing one of the 6 atoms (with equal probability), and flipping it. For example, we might choose to flip the second atom:

$$011001 \quad \overset{\text{flip second atom}}{\longrightarrow} \quad 001001 \qquad (q = 1/6). \tag{13.1.7}$$

If we start from the other state, the reverse process has the same probability:

$$001001 \quad \overset{\text{flip second atom}}{\longrightarrow} \quad 011001 \qquad (q = 1/6). \tag{13.1.8}$$

Hence, this scheme for walking through the "state space" is said to be *unbiased*. Note that, for a given walking scheme, it is not always possible to connect every two states by a single step; for example, in this case we can't go from 000000 to 111111 in one step.

There is more than one possible walking scheme; for instance, a different scheme could involve randomly choosing *two* atoms and flipping them. Whatever scheme we choose, however, the most important thing is that the walk must be unbiased: each possible step must occur with the same probability as the reverse step. Otherwise, the above proof that the Metropolis algorithm satisfies detailed balance would not work.

13.2: The Ising Model

13.2.1 Problem Statement

To better understand the above general formulation of the MCMC method, let us apply it to the **2D Ising model**, a simple and instructive model which is commonly used to teach statistical mechanics concepts. The system is described by a set of N "spins", arranged in a 2D square lattice, where the value of each spin S_n is either $+1$ (spin up) or -1 (spin down). This describes a hypothetical two-dimensional magnetic material, where the magnetization of each atom is constrained to point either up or down.

Each state can be described by a grid of $+1/=1$ values. For example, for a 4×4 grid, a typical state can be represented as

$$\{S\} = \begin{Bmatrix} +1 & +1 & +1 & -1 \\ +1 & -1 & -1 & +1 \\ -1 & +1 & +1 & -1 \\ -1 & +1 & -1 & -1 \end{Bmatrix}, \tag{13.2.1}$$

and the total number of possible states is $2^{16} = 65536$.

The energy of each state is given by

$$E(\{S\}) = -J\sum_{\langle ij\rangle} S_i S_j, \tag{13.2.2}$$

where $\langle ij\rangle$ denotes pairs of spins, on adjacent sites labeled i and j, which are adjacent to each other on the grid (without double-counting). We'll assume periodic boundary conditions at the edges of the lattice. Thus, for example,

$$\begin{Bmatrix} +1 & +1 & +1 & +1 \\ +1 & +1 & +1 & +1 \\ +1 & +1 & +1 & +1 \\ +1 & +1 & +1 & +1 \end{Bmatrix} \quad E = -32J. \tag{13.2.3}$$

$$\begin{Bmatrix} +1 & +1 & -1 & +1 \\ +1 & +1 & +1 & +1 \\ +1 & +1 & +1 & +1 \\ +1 & +1 & +1 & +1 \end{Bmatrix} \quad E = -24J. \tag{13.2.4}$$

For each state, we can compute various quantities of interest, such as the mean spin

$$S_{\text{avg}}(\{S\}) = \frac{1}{N}\sum_i S_i. \tag{13.2.5}$$

Here, avg denotes the average over the lattice, for a given spin configuration. We are then interested in the thermodynamic average $\langle S_{\text{avg}}\rangle$, which is obtained by averaging S_{avg} over a thermodynamic ensemble of spin configurations:

$$\langle S_{\text{avg}}\rangle \;=\; \sum_{\text{possible states } \{S\}} S_{\text{avg}}(\{S\})\,\pi(\{S\}), \tag{13.2.6}$$

where $\pi(\{S\})$ denotes the probability of a spin configuration:

$$\pi(\{S\}) = \frac{1}{Z}\exp\left(-\frac{E(\{S\})}{kT}\right). \tag{13.2.7}$$

13.2.2 Metropolis Monte Carlo Simulation

To apply the MCMC method, we design a Markov process using the Metropolis algorithm discussed above. In the context of the Ising model, the steps are as follows:

1. On step k, randomly choose one of the spins, i, and consider a candidate move which consists of flipping that spin: $S_i \to -S_i$.
2. Calculate the change in energy that would result from flipping spin i, relative to kT, i.e. the quantity:

$$Q \equiv \frac{\Delta E}{kT} = -\left[\frac{J}{kT}\sum_{j \text{ next to } i} S_j\right]\Delta S_i, \tag{13.2.8}$$

where ΔS_i is the change in S_i due to the spin-flip, which is -2 if $S_i = 1$ currently, and $+2$ if $S_i = -1$ currently. (The reason we calculate $Q \equiv \Delta E/kT$, rather than ΔE, is to keep the quantities in our program dimensionless, and to avoid dealing with very large or very small floating-point numbers. Note also that we can do this calculation without summing over the entire lattice; we only need to find the values of the spins adjacent to the spin we are considering flipping.)

- If $Q \leq 0$, *accept* the spin-flip.
- If $Q > 0$, *accept* the spin-flip with probability $\exp(-Q)$. Otherwise, *reject* the flip.

3. This tells us the state on step $k+1$ of the Markov chain (whether the spin was flipped, or remained as it was). Use this to update our moving average of S_{avg} (or whatever other average we're interested in).
4. Repeat.

The MCMC method consists of repeatedly applying the above Markov process, starting from some initial state. We can choose either a "perfectly ordered" initial state, where $S_i = +1$ for all spins, or a "perfectly disordered" state, where each S_i is assigned either $+1$ or -1 randomly.

In some systems, the choice of initial state is relatively unimportant; you can choose whatever you want, and leave it to the Markov chain to reach the stationary distribution. For the Ising model, however, there is a practical reason to prefer a "perfectly ordered" initial state, for the following reason. Depending on the value of J/kT, the Ising model either settles into a "ferromagnetic" phase where the spins are mostly aligned, or a "paramagnetic" phase where the spins are mostly random. If the model is in the paramagnetic phase and you start with an ordered (ferromagnetic) initial state, it is easy for the spin lattice to "melt" into disordered states by flipping individual spins, as shown in Figure 13.2.1:

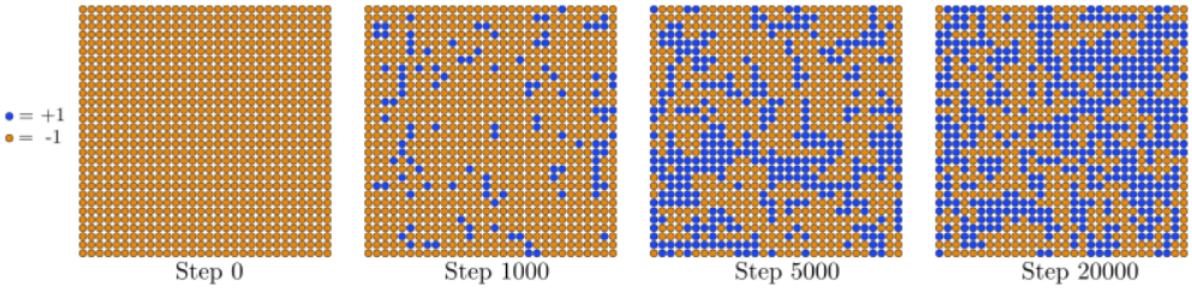

Figure 13.2.1: Progress of a Monte Carlo simulation of an Ising model with an ordered initial state, for a 30×30 lattice with $J/kT = 0.25$. The ordered spin lattice "melts" into a disordered configuration, which is the thermodynamic equilibrium for this value of $J/kT = 0.25$.

In the ferromagnetic phase, however, if you start with a disordered initial state, the spin lattice will "freeze" by aligning adjacent spins. When this happens, large domains with opposite spins will form, as shown in Figure 13.2.2 These separate domains cannot be easily aligned by flipping individual spins, and as a result the Markov chain gets "trapped" in this part of the state space for a long time, failing to access the more energetically favorable set of states where most of the spins form a single aligned domain. (The simulation will eventually get unstuck, but only if you wait a *very* long time.) The presence of domains will bias the calculation of S_{avg}, because the spins in different domains will cancel out. Hence, in this situation is better to start the MCMC simulation in an ordered state.

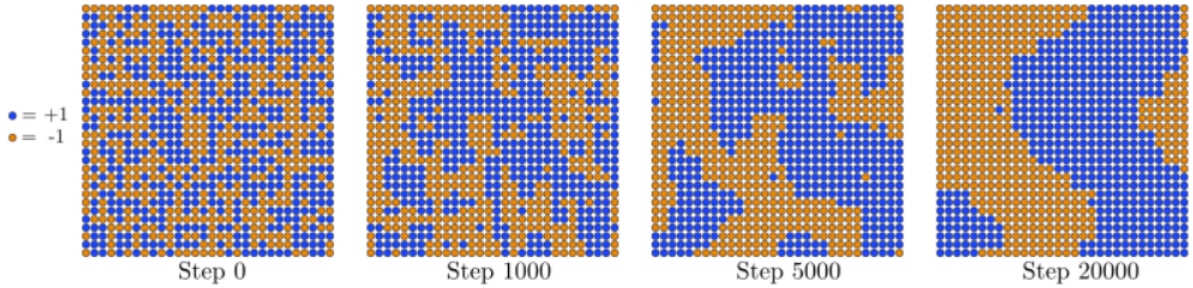

Figure 13.2.2: Progress of a Monte Carlo simulation of an Ising model with a disordered initial configuration, for a 30×30 lattice with $J/kT = 1$. As the disordered spin lattice "freezes", it forms long-lasting domains which can interfere with calculations of S_{avg}.

Index

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: Computational Physics (Chong)

Webpages: 67

All licenses found:

- CC BY-SA 4.0: 86.6% (58 pages)
- Undeclared: 13.4% (9 pages)

By Page

- Glossary - *Undeclared*
- Detailed Licensing - *Undeclared*