Introduction to Computational Physics Lecture Notes

James Farrell and Jure Dobnikar

Spring 2023

Contents

Part I

TOOLS

Chapter 1

Programming with Python

1.1 Introduction

The first chapter of this part is a brief introduction to programming in the Python language. Without an understanding of how to program computers, the rest of the information in this book is quite useless! We encourage you to take this chapter very seriously. Whether you are completely new to programming, or have a lot of experience, you should find something of value here. The skills you will begin to learn in this chapter have applications not only in physics, but also in science more generally, and in other fields besides.

Many books, introductory and advanced, have been written on this topic. We will take a pragmatic approach, focusing on the ideas that will be crucial to completing the exercises in this course. For an introduction to Python programming set in a broader context, we encourage you to read A Byte of Python, a free, online tutorial aimed at total beginners, which is also available in a Mandarin translation.

1.1.1 What is Python?

Python is a programming language—a way of telling a computer what to do. In particular, Python is an **interpreted**, **object-oriented**, **high-level** programming language.

An **interpreted** language is a programming language whose programs are translated into machine code by an interpreter at run-time—this is opposed to a **compiled** language, whose programs are translated into machine code *before* run-time.

Object-oriented programming is a **programming paradigm** based on the concept of "objects", which can contain data, in the form of fields (often known as attributes), and code, in the form of procedures (often known as methods). Actually, programs written in Python are not restricted to the object-oriented paradigm—Python supports **many programming paradigms**, such as object-oriented, imperative, procedural, and functional programming, to greater or lesser degree. More information on programming paradigms can be found here.

A **high-level** language is one with strong abstraction from the details of the computer. Programs written in high-level languages are usually easier to understand and modify than their low-level counterparts, and hide from the programmer the fine details of how computers actually work (the details are abstracted away), such as memory management. Programs written in **low-level** languages also have advantages; for example, the programmer can write programs tailored to the specific machine on which the programs are to be run, resulting in more efficient code.

As such, Python is a great first language for a new programmer: she need not worry about the details of compilation, as her programs are **interpreted** when run; she may explore many **programming paradigms**, and learn which paradigm is appropriate for which task; and from the very beginning she may write programs using using complex structures, written with a syntax similar to natural language, owing to Python being a **high-level** language. However, her programs will not be as efficient as those written in other languages, whose programs interface directly with hardware (**low-level languages**), or can be optimized at compilation time (**compiled languages**).

1.1.2 Installing Python

In order to complete this course, you will need two things: a working Python installation; an integrated development environment (IDE). If you understand all of the terms in the preceding sentence, you are probably ready to begin. If not, I recommend you use the `Conda` package and environment management system and the `PyCharm` IDE.

`Conda`

Instructions on how to install `Conda` on Windows, Mac, or Ubuntu. Choose the Python 3 version.

`PyCharm`

Instructions on how to install `PyCharm`.

1.2 Basics I

By the end of this section, you should be able to understand and write python code that looks like the following:

```
def factorial(n: int) -> int:  # function definition statement
    """

    evaluates n! = n * (n - 1) * ... * 2 * 1
    0! evaluates to 1

    >>> factorial(0)
    1

    >>> factorial(10)
    3628800

    >>> factorial(-1)
    Traceback (most recent call last):
    ValueError: n! is undefined for n less than zero

    >>> factorial(3.141)
    Traceback (most recent call last):
    TypeError: n is not an integer

    :param n: element of the factorial sequence to be evaluated
    :return: n!
    """

    if n < 0:  # if statement
        raise ValueError("n! is undefined for n less than zero")  # raise statement
    elif not isinstance(n, int):
        raise TypeError("n is not an integer")  # raise statement

    n_factorial = 1  # assignment statement

    while n > 1:  # while statement
        n_factorial = n_factorial * n  # assignment statement
        n = n - 1  # assignment statement

    return n_factorial  # return statement
```

As you might have guessed, this is program that evaluates the factorial of a number. We'll go through every element one step at a time.

Tip The code elements `## function definition statement`, `## if statement`, etc are *comments*. Comments are ignored when the code is run, but provide useful information about the code. You should use comments liberally! In the python language, anything following a `##` that is not part of a *string literal* is parsed as a comment. Generally, comments should be more informative than these examples; for example, it is quite obvious (to the initiated) that line 1 is a function definition statement. Comments should explain your intent, not simply reproduce what is obvious from the code.

The core functionality is contained in lines 32–36:

```
32      while n > 1:  # while statement
33          n_factorial = n_factorial * n  # assignment statement
34          n = n - 1  # assignment statement
35
36      return n_factorial  # return statement
```

Let's take a closer look at line 32.

1.2.1 Assignment statements, names, expressions

Line 32 is an *assignment statement*. *Statements* are sections of code that *do* something. Assignment statements are the most fundamental statements in the language - they allow you to store the result of a calculation in a variable.

On the left-hand-side (LHS) of the assignment statement, there should be a *name*, or *identifier*. In this instance, a name is what you might refer to as a variable. In our example, the name is `n_factorial`. There are some rules that determine whether a name is valid. The left-hand-side of an assignment statement must be a *valid name*.

- names can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
- names like `myClass`, `var_1` and `print_this_to_screen`, all are valid.
- a name cannot start with a digit.
- `1variable` is invalid, but `variable1` is perfectly fine.
- it is unwise to use the names of built-in functions (`print`, `list`, `input`, etc) as names.
- it is **illegal** to use *reserved words*, or *keywords* (`False`, `for`, `return`, etc) as names.

Tip Names should be chosen to maximise code readability. Avoid single letter names (e.g. `i, j, x, y`) where a more informative choice can be made (e.g. `height, weight, number_of_people`). Follow the conventions in the style guide.

- function and variable names should be `lower_case_with_underscores`
- class names should be `CamelCase`
- names for constants should be `UPPER_CASE_WITH_UNDERSCORES`
- module names should be short and all lowercase

The right-hand-side (RHS) of the assignment statement must be a *valid expression*. Expressions can be made up of *literals*, *names*, *operators*, and *function calls*.

- `1` and `"hello world"` are literals, and valid expressions
- `2+3` and `print(2+3)` are valid expressions
- `1+`, `/2`, `"string` are not expressions and will cause a syntax error
- "syntax" errors, errors in *grammar*, occur when you have written something that the interpreter cannot understand, e.g., an invalid expression,

```
>>> 1 +
  File "<stdin>", line 1
    1 +
      ^
SyntaxError: invalid syntax
```

(The code block above is a python console session. Lines beginning with `>>>` are typed by you; lines without are the console output.)

When the assignment statement is executed,

- the expression on the RHS is evaluated
- the name on the LHS is *bound* to the result

The name then behaves like that result in subsequent expressions and statements.

```
>>> n_factorial = 1  # assignment statement
>>> n_factoral  # expression statement
1
```

Tip In the previous console session, the second line is an *expression statement*. This a statement made that is just an expression. If you type an expression statement into a console session, and the expression evaluates to something other than `None`, the result will be printed to the screen. This only works in console sessions; an expression statement in a `\*.py` file will be executed, but it's value will **not** be printed to the screen.

1.2.2 Literals, types, operators

The simplest expressions are *literals*. Literals are notations for constant values of some built-in *type*. The type of an object determines the result of applying *operators* to it.

```
>>> 1 + 2
3
```

In the above console session, `1` and `2` are *integer literals*. Their type is `int`, and their values are the integers 1 and 2. `+` is a *binary operator*. When the types of the objects on the LHS and RHS of the `+` are `int`, integer addition will be performed.

Arithmetic operators

The arithmetic operators are

- `+` performs addition
- `-` performs subtraction
- `*` performs multplication
- `//` performs integer division
- `/` performs float division
- `%` performs modulo
- `**` performs exponentiation
- `@` performs matrix multiplication (Python 3.6+)

As a *unary operator*, `-` in the expression `-x` multiplies `x` by -1. For all of these operators except float division and exponentiation, if the LHS and RHS are both `int`, the expression `LHS ∘ RHS` evaluates to an `int`.

Numeric types

As well as the `int` type for representing integers, there are the `float` type, for representing floating point numbers (real numbers, with some caveats) and the `complex` type, for representing complex numbers. Floats can be entered in decimal or exponential format. Imaginary numbers are entered the same way, but with a letter `j` at the end. Complex numbers are entered as a sum of a float and an imaginary number.

```
 >>> 1000.0 # float
1000.0
>>> 1e3
1000.0
>>> 1e20
1e+20
>>> 1e-20
1e-20
>>> 2.5j  # complex
2.5j
>>> 1e10 + 1e-10j  # complex
(10000000000+1e-10j)
```

When the LHS and RHS are not the same numeric type, the following *arithmetic conversion* rules are observed:

- if either argument is a complex number, the other is converted to complex
- otherwise, if either argument is a float, the other is converted to float
- otherwise, both must be integers and no conversion is necessary

What you might consider a fourth numeric type is the `bool` type for boolean numbers. A `bool` is equal to either `True` or `False`. In arithmetic expressions, `bool` objects are treated as 0 or 1, for `False` and `True`, respectively. The type of a literal, name, or other object can be discovered by using the `type` function,

```
>>> type(True)
<class 'bool'>
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type(1.0j)
<class 'complex'>
```

Tip Since Python 3, the `/` operator performs float division, regardless of whether both the LHS and RHS are `int`. Take note of the following examples.

```
>>> 1 // 2
0
>>> 1 / 2
0.5
>>> 2 / 1
2.0
```

Time to go back to our `factorial` code.

```
33          n_factorial = n_factorial * n  # assignment statement
34          n = n - 1  # assignment statement
```

Lines 33 and 34 are assignment statements with arithmetic expressions on the RHS. You will notice in line 33 that `n_factorial` appears on both sides of the `=` sign. This is not a problem, as the RHS is evaluated first, then the LHS is about to that value. In variable terms, the old value of `n_factorial` is replaced by `n_factorial * n`.

To understand line 32, we have to familiarise ourselves with *comparison operators*, and *while statements*—a kind of *control flow*.

Comparison operators

As well as arithmetic operators, the Python language uses *comparison operators*. Comparison operators compare two expressions, and evaluate to either `True` or `False`. Among the comparison operators are

- `==` evaluates to `True` if the LHS and RHS are equal, `False` otherwise
- `!=` is the negation of `==`
- `<` and `<=` evaluate to `True` if the LHS is less than, or less than or equal to the RHS
- `>` and `>=` evaluate to `True` if the LHS is greater than, or greater than or equal to the RHS
- `is` evaluates to `True` if the LHS and RHS are *identical objects*, `False` otherwise
- `is not` is the negation of `is`

Info: The distinction between `is` (identical) and `==` (equal) is a subtle one. Consider the following examples.

```
>>> 1 == 1.0  # 1 is converted to a float
True
>>> 1 is 1.0  # 1 is an int literal, 1.0 is a float literal
False
>>> x = 1
>>> y = 1
>>> x == y  # x and y are both equal to 1
True
>>> x is y  # x and y are both int with value 1
True
```

Besides arithmetic and comparison operators, there are also boolean operators (for use with `bool` arguments) and bitwise operators (for use with `int` arguments).

```
32      while n > 1:  # while statement
```

In line 32 of our `factorial` code, the expression `n > 1` evaluates to `True` if `n` is greater than `1`, and `False` otherwise.

1.2.3 The `while` statement (indefinite iteration)

In many situations, we want the behaviour of our program to depend on the current properties of a variable, or that of some expression containing it. For example, we may want to *loop* over a set of statements until some condition obtains; or perhaps we would like our program to treat negative integers and positive integers differently. We can achieve these aims by controlling the flow of execution of our program with *control flow statements*.

One such control flow statement is the `while` statement. The `while` statement has the form

```
1 while expr:
2     suite
3 other code
```

When a `while` statement is encountered, the expression `expr` is evaluated. If `expr` evaluates to an object that is considered *false*, then line 2 is ignored, and execution moves to line 3. If `expr` evaluates to an object that is considered *true*, then the set of statements in the block `suite` is executed. After executing `suite`, the *truth value* of `expr` is tested again. This *loop* continues until `expr` evaluates to a *false* object.

Notice the *indentation* on line 2. While lines 1 and 3 begin at the same horizontal position, line 2 is *indented* by 4 spaces. The content of the suite is defined by those statements immediately following the `while` statement at a greater level of indentation. When a statement with the same indentation or lesser is encountered, it is not executed, and execution returns to the `while` statement.

Consider the following console session that calculates the sum of the integers less than or equal to 5.

```
>>> total = 0
>>> n = 5
>>>
>>> while n > 0:
...     total = total + n
...     n = n - 1
...
>>> total
15
>>> n
0
```

When we first encounter line four, the value of `n` is `5`. `5 > 0` evaluates to `True`, which is a true value, so the statements on lines five and six are executed. The value of `total` is updated to `5` (line five), and the value of `n` is updated to `4` (line six). We then return to line four. `4 > 0` still evaluates to a true value, so lines five and six are executed once more. After lines five and six have been executed a total of five times, the value of `total` is 15, and the value of `n` is `0`. `0 > 0` evaluates to `False`, which is a false value, so execution moves to line eight, and we escape the loop.

What is a true or false object? For the purposes of truth-value testing,

- the constants `None` and `False` are defined to be false
- zero of any numeric type is defined to be false: `0, 0.0, 0.0e10, 0j, Decimal(0), Fraction(0, 1)`
- empty sequences and collections are also false: `'', (,), [], {}, set(), range(0)`

As a consequence, `while True:`, `while 1:`, and `while 3.142:` are functionally equivalent.

Returning to our initial problem,

```
33          n_factorial = n_factorial * n  # assignment statement
34          n = n - 1  # assignment statement
```

we can see that lines 33 and 34 will be executed a total of `n - 1` times, with the variable `n_factorial` multiplied by every integer `m` for `1 < m < n`.

Question 1 The Fibonacci Sequence: part 1

You are likely familiar with the Fibonacci sequence:

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0;\ F_1 = 1$$

In the limit of large n, the ratio F_n/F_{n-1} tends to the golden ratio, ϕ:

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \phi = \frac{1+\sqrt{5}}{2}$$

Using only the Python we have learned so far, [a] find the smallest n such that:

$$\left\| \frac{F_n}{F_{n-1}} - \phi \right\| < 10^{-10}$$

[a] you may also need to use the built-in `abs` function; `abs(x)` returns the absolute value of `x`.

1.2.4 Functions and modules

Consider the following console session.

```
>>> print(888)
888
```

`print` is a *function*. It's purpose is to take it's *argument* (`888`) and print it to the screen. We pass the argument to the function *via* the *function call operator*, `()`.

- generally speaking, functions can take any number of arguments (including none)
- functions *return* some object (the result of the operation of the function on its arguments)
- if an expression is a function call, it evaluates to the function's return value
- the return value of `print` is always `None`

Python has many built-in functions, such as `print`, that are always available. Other functions can be *imported* from *modules*. For mathematical functions other than basic arithmetic (i.e., `+, -, *, /, **`), we can import the math module with an *import statement*,

```
>>> import math # import statement
>>> math.sqrt(4.0) # square root of 4
2.0
>>> math.sqrt(2.0) # square root of 2
1.4142135623730951
>>> math.acos(1.0) # arccos of 1
0.0
```

The notation `module.object` refers to the attribute `object` of `module`. The attribute could be a constant, a function, a class, or another module. In the above examples, we access the `sqrt` function of the `math` module. As well as importing functions from `math`, we can import constants,

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
```

Having to retype `math` every time is tedious, and makes the code less readable. Instead, we can import math with an *alias*,

```
>>> import math as m
>>> m.asin(1) # arcsin of 1
1.5707963267948966
```

or import only the *names* we want to use,

```
>>> from math import atan
>>> atan(1.0) # arctan of 1
0.7853981633974483
```

The `def` and `return` statements

We can define our own functions. A function definition has the following form:

```
1 def function_name(argument_list):  # function definition statement
2     suite  # function body
3     return return_list  # return statement, optional
```

Unlike a `while` statement, when the `def` statement is encounted, the suite is not executed. Rather, the suite is checked for syntax errors, and then a *function object* is created. When the function is called with the function call operator, *i.e.* `function_name()`, the statements in `suite` are executed, and `function_name()` evaluates to `return_list`. If `return_list` is not provided, or the `return statement` is omitted altogether, the function call evaluates to `None`.

The `suite` is defined by the continuous block of statements following the `def` statement that are at a greater level of indentation that the `def` statement itself (just like we saw for the `suite` in the `while` statement).

We can import our own functions in the same way we import functions from modules.

```
# square.py

def square(x: float) -> float:
    return x * x
```

```
# script.py

from square import square

x = 4
x_squared = square(x)
print(x)
```

Functions can be written with two types of arguments:

- *positional* arguments, that have no default value, and must by explicitly provided to a function call
- *keyword* arguments, that have some default value, and may be omitted from a function call

Consider a function that takes two numbers, x and n, and returns the n^{th} power of x. We give n a default value 1. The correct way to construct the `def` statement is as in the following:

```
# power.py

def power(x: float, n: int=1):
return x ** n
```

Positional arguments are specified first, followed by keyword arguments. The `def` statement

```
def power(n=1, x):
    return x ** n
```

is a syntax error.

The rules for function calls are somewhat more lax,

```
>>> from power import power
>>>
>>> a, b = 2, 3
>>> power(x=a)  # n can be omitted from the argument list
2
>>>
>>> power(n=a) # x cannot be omitted from the argument list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'x'
>>>
>>> power(x=a, n=b) # x can be supplied as a keyword argument
8
```

```
>>>
>>> power(n=a, x=b) # if all arguments are supplied as keyword arguments, the order is arbitrary
9
>>>
>>> power(a, b) # if all arguments are supplied as positional arguments,
8
>>>
>>> power(b, a) # the order is critical
9
>>>
>>> power(n=a, x) # positional arguments cannot follow keyword arguments
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

We recommend you follow the pattern in lines 3 and 9 to maximise the readability of your code and avoid errors.

Type hints

In statically-typed languages, such as FORTRAN and C, the types of objects and arguments to functions are fixed as or before they are used. Python is by contrast a dynamically-typed language; functions in Python do not require arguments of particular types, and the types of arguments are not checked by the interpreter at run time. Additionally, an identifier may refer to objects of different types over the course of its lifetime. Nevertheless, it is useful to annotate Python code with *type hints*, indicating the types or behaviours that are expected in a particular situation. In the function definition statement of the `factorial` function,

```
1 def factorial(n: int) -> int:  # function definition statement
```

type hints are used to indicate that both the type of the argument `n`, and the return type are both `int`. Even with type hints, the types of arguments are never checked unless we explicitly require it (*e.g.,* on line 27 of the same code), but provide useful information to the user and the IDE, as well as appearing in the output of the `help` function.

docstrings and doctests

The syntax of lines 1 and 36 in `factorial` are now clear to us. We now turn to the largest section of the code. 22 of the 36 lines, from lines 2 to 23, are devoted to *documentation*. Inline comments (`## like this`) help to elucidate small blocks of statements, but *docstrings* draw the bigger picture.

```
2     """
3
4     evaluates n! = n * (n - 1) * ... * 2 * 1
5     0! evaluates to 1
6
7     >>> factorial(0)
8     1
9
10    >>> factorial(10)
11    3628800
12
13    >>> factorial(-1)
14    Traceback (most recent call last):
15    ValueError: n! is undefined for n less than zero
16
17    >>> factorial(3.141)
18    Traceback (most recent call last):
19    TypeError: n is not an integer
20
21    :param n: element of the factorial sequence to be evaluated
```

```
22      :return: n!
23      """
```

A docstring immediately follows a `def` statement and is enclosed in triple double quotes. By convention, the first few lines (here, lines 4–5) describe in words the problem that the function solves. It should provide a sufficiently detailed explanation for a user to know whether it is the function they are looking for.

The following block of lines contain usage examples in the form of transcribed console sessions (lines 7–19). These lines a more than just hints—we will come back to them shortly.

The last block of lines should contain details about the parameters and return value (lines 21–22). From line 1, we can see the author intended for the `factorial` function to take an integer parameter, `n`. This comment does not imply if you give the function a non-integer value then an error will be thrown (lines 17–19, however, state this explicitly); rather it implies that problem-free execution is not *guaranteed*. An explicit error is really the best-case scenario; in the worst-case scenario, a function returns a nonsense value, execution proceeds silently, and the program returns an incorrect result *that you may not even be able to identify as such!*

A function's docstring is stored in its `__doc__` attribute.

```
>>> from factorial import factorial
>>> print(factorial.__doc__)

    evaluates n! = n * (n - 1) * ... * 2 * 1
    0! evaluates to 1

    >>> factorial(0)
    1

    >>> factorial(10)
    3628800

    >>> factorial(-1)
    Traceback (most recent call last):
    ValueError: n! is undefined for n less than zero

    >>> factorial(3.141)
    Traceback (most recent call last):
    TypeError: n is not an integer

    :param n: element of the factorial sequence to be evaluated

    :return: n!

>>>
```

In a console session, the `help` function, when supplied with an object as its argument, will bring up a page of useful information of which the docstring forms part.

Now back to those usage examples (lines 7–19); in addition to being handy hints, they form a suite of tests. The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. If your tests fail, then either

a. your docstring needs to be updated, or
b. your code no longer works properly.

`doctest` can be invoked in a terminal session,

```
farrelljd@sommerfugl:~/2019/programming_course$ python3 -m doctest factorial.py
farrelljd@sommerfugl:~/2019/programming_course$
farrelljd@sommerfugl:~/2019/programming_course$ python3 -m doctest -v factorial.py
Trying:
    factorial(0)
Expecting:
```

```
    1
ok
Trying:
    factorial(10)
Expecting:
    3628800
ok
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
    ValueError: n! is undefined for n less than zero
ok
Trying:
    factorial(3.141)
Expecting:
    Traceback (most recent call last):
    TypeError: n is not an integer
ok
1 items had no tests:
    factorial
1 items passed all tests:
   4 tests in factorial.factorial
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
farrelljd@sommerfugl:~/2019/programming_course$
```

where the `-v` flag enables verbose output. In a PyCharm session, right-click on your code and choose *Run 'Doctest <module>'*.

Writing good documentation may seem like a hassle now, but it saves a lot of time in the long run. Best practice is to write a docstring and doctests *before* you implement your function! That way, you have a clear statement of how your function should behave in a variety of circumstances, and can periodically check your progress with the doctest module.

1.2.5 The `if` statement

`while` statements allow us to repeatedly execute blocks of code while some condition is satisfied. `if` statements allow use to execute different blocks of code depending on the truth value(s) of a set of expressions.

The `if` statement has the form:

```
1  if expression1:
2      suite1
3  elif expression2:
4      suite2
5  ...
6  elif expressionY:
7      suiteY
8  else:
9      suiteZ
10 other code
```

The flow is simple: if `expression1` evaluates to a true value, `suite1` is executed, and the interpreter moves to the end of the `if` block (line 10). Otherwise, the truth value of `expression2` is evaluated; if it is true, `suite2` is executed, and the interpreter moves to the end of the `if` block. All expressions are evaluated until a true value is obtained, whereupon the corresponding suite is executed, and all subsequent expressions ignored. If no expression evaluates to a true value, the suite following the `else` statement is executed.

An `if` block has exactly one `if` statement, zero or more `elif` statements, and zero or one `else` statement.

continue and break

Complex control flow can be implemented by combining `if` and `while` statements. Two useful statements for use within `while` (and `for`) blocks are `continue` and `break`. When a `continue` statement is encountered in a suite, the remainder of the suite is skipped, and execution resumes at the `while` statement. When a `break` statement is encountered in a suite, the remainder of the suite is skipped, and execution resumes *at the end* of the `while` block. The `else` clause of the `while` block, if present, **is not executed**.

```
>>> i = 0
>>> while True:  # loop forever
...     i += 1
...     if i % 2 == 0:  # skip even numbers
...             continue
...     elif i % 5 == 0:  # --> print odd numbers less than five
...             break
...     print(i)
... else:
...     print("this line is never executed")
...
1
3
>>>
```

Returning to the factorial code,

```
25      if n < 0:  # if statement
26          raise ValueError("n! is undefined for n less than zero")  # raise statement
27      elif not isinstance(n, int):
28          raise TypeError("n is not an integer")  # raise statement
```

line 25 tests whether `n` is less than zero. If true, line 26 is executed. Otherwise, line 27 tests whether `n` is an not instance of the type `int` by means of the built-in `isinstance` function. If true, line 28 is executed.

1.2.6 Exception handling

What should we do if the value supplied to our `factorial` function is less than zero? If we do nothing, the `while` suite is skipped, and, thanks to line 30, the return value for all `n < 0` is 1.

This is a problem. Somewhere, some piece of code has called the `factorial` function with a negative number as its argument, and nobody noticed. In all likelihood, that number is negative because of *an error somewhere else in the code*. We would like to find that error *as soon as possible*. We can do this by *raising an exception*. An exception is not a Python error, but something we should *take exception to*, *i.e.* an outcome we should reject.

A `raise` statement has the form

```
1  raise exception_class("error_message")
```

When `raise` statement is encountered, a *traceback* is printed to the screen, which tells us the sequence of statements that brought us to the `raise` statement, along with function names and line numbers. Finally, the name of the `error_class` and the `error_message` are printed to the screen. The traceback helps find the source of the error, while the error class and message tell us what the error is. The error message is of `string` type—a type for things best represented with words and letters. We will discuss strings in detail later. For now, be aware that a string is a concatentation of characters and whitespace enclosed in either single quotes `''` or double quotes `""`.

```
25      if n < 0:  # if statement
26          raise ValueError("n! is undefined for n less than zero")  # raise statement
```

```
1   # some_code.py
2   from factorial import factorial
3
4   factorial(-1)
```

```
farrelljd@sommerfugl:~/2019/programming_course$ python some_code.py
Traceback (most recent call last):
  File "some_code.py", line 4, in <module>
    factorial(-1)
  File "/home/farrelljd/2019/programming_course/factorial.py", line 29, in factorial
    raise ValueError("n! is undefined for n less than zero")  # raise statement
ValueError: n! is undefined for n less than zero
farrelljd@sommerfugl:~/2019/programming_course$
```

From the traceback, we find that the problem arises from supplying a negative value to `factorial` in line 4 of *some_code.py*.

From the list of built-in exceptions, we find that a `ValueError` is raised when

> "... a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`."

which seems appropriate to our case. If no built-in exception adequately describes the situation, it is possible to define a new one.

```
27      elif not isinstance(n, int):
28          raise TypeError("n is not an integer")  # raise statement
```

A `TypeError` is raised when

> "... an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
>
> This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, NotImplementedError is the proper exception to raise.
>
> Passing arguments of the wrong type (e.g. passing a list when an int is expected) should result in a TypeError, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a ValueError."

which, again, seems to be the right choice.

And that's it for the `factorial` function!

Question 2 The Fibonacci Sequence: part 2

Write a documentation string for your Fibonacci sequence implementation.

I include a code skeleton below, with a sample docstring. The docstring is not complete!

```
def fibonacci(n: int):
    """

    calculates the n^th value of the Fibonacci sequence, F_n

    >>> fibonacci(0)
    0

    >>> fibonacci(1)
    1

    >>> fibonacci(100)
    354224848179261915075

    :param n: element of the Fibonacci sequence to be calculated
    :type n: int

    :return: F_n
    """

    return
```

1.3 Basics II

By the end of this section, you should be able to understand and write python code that looks like the following:

ljpotential.py:

```
def lennard_jones_potential(r: float, epsilon: float = 1.0, sigma: float = 1.0):
    """

    Calculates the Lennard-Jones potential for particles with diameter sigma
    at a separation r with a well-depth epsilon,

        V_{LJ}(r) = 4 epsilon ( (sigma/r)^{12 - (sigma/r)^6 ) .

    >>> lennard_jones_potential(1.0, 1.0, 1.0)
    0.0

    >>> lennard_jones_potential(2**(1/6), 1.0, 1.0)
    -1.0

    >>> lennard_jones_potential(0.0, 1.0, 1.0)
    Traceback (most recent call last):
    ZeroDivisionError: float division by zero

    >>> lennard_jones_potential(-1.0, 1.0, 1.0)
    Traceback (most recent call last):
    ValueError: distance between particles is negative

    >>> lennard_jones_potential(1.0, -1.0, 1.0)
    Traceback (most recent call last):
    ValueError: particle diameter is not strictly positive

    """

    if r < 0.0:
        raise ValueError("distance between particles is negative")
    elif sigma <= 0.0:
        raise ValueError("particle diameter is not strictly positive")

    r6 = (sigma / r) ** 6

    return 4 * epsilon * r6 * (r6 - 1)
```

pairpotential.py:

```
from itertools import combinations
from math import sqrt
from typing import Callable


def pair_potential(xs: [[float]], potential: Callable[[float, ...], float],
                   potential_args: tuple = ()) -> float:

    """

    Calculates the potential energy of configuration of particles.

    >>> from ljpotential import lennard_jones_potential as lj

```

```
    >>> pair_potential(x=[[0.0,0.0,0.0]], potential=lj)
    0.0

    >>> pair_potential(x=[[0.0,0.0,0.0],[0.0,0.0,1.0]], potential=lj)
    0.0

    >>> pair_potential(x=[[0.0,0.0],[0.0,1.0],[0.0,2.0]],
    ... potential=lj,
    ... potential_args=(1.0, 1.0)) # 2D configuration
    -0.0615234375

    :param x: positions of the particles
    :param potential: computes the energy of one pair; must be of the form f(x, *args)
    :param potential_args: arguments to pass to the function

    :return: energy of the configuration
    :rtype: float
    """

    energy = 0.0

    for x1, x2 in combinations(x, 2):  # for statement
        r_squared = 0.0
        for c1, c2 in zip(x1, x2):  # for statement
            r_squared += (c1 - c2) * (c1 - c2)
        r = sqrt(r_squared)  # sqrt imported from math module
        energy += potential(r, *potential_args)

    return energy
```

ljpotential.py contains a function that evaluates the Lennard-Jones potential between two particles at some distance `r`. The Lennard-Jones potential is given by the expression:

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

where σ is the diameter of the particles and ϵ is the depth of the potential well. It finds use as a simple model for pair interactions in noble gases and simple fluids, among other applications. We are already able to fully understand the contents of *ljpotential.py* with the Python from §1.2.

pairpotential.py contains a function that calculates the potential energy of a collection of particles that interact with potential `potential`. The energy of a collection of particles in which only pair-wise interactions occur is given by

$$V_{\text{total}} = \sum_{i \neq j} V(r_{ij})$$

where V is a pair-wise potential energy function, i and j are indices of particles, and r_{ij} is the distance between the particle centres of mass. The Python function `pair_potential` is designed to be agnostic of the specific nature of the interaction, and so can be used in many situations.

The idea of separating out functionality in this way, making code *modular*, is very important regardless of the language used, and can speed up writing, reading, changing, and debugging code dramatically.

1.3.1 Lists

The documentation string of `pair_potential` identifies the variable `x`, the positions of the particles, as a *list of lists*. What is a list of lists? A list whose elements are lists, of course.

In many situations, we want to represent collections of objects in our code. Two examples are the coordinates of a physical system, and a database of customers and their details. We are interested not only in representing the basic elements themselves (the x-coordinate of a particle, which might be represented with a float, or the name of an individual customer, which might be represented with a string), but also

the relationships between them (which x-coordinate is paired with which y-coordinate, which name goes with which phone number). In Python, the job of representing these relationships is done by *collections*, or *container datatypes*.

Different containers are optimised for different tasks. We will not go into any low-level detail about how containers work; we will just cover the basics of how the built-in containers are applied. The curious can read more about other Python containers here.

The aforementioned *list* is a built-in container datatype. A list may contain any number of elements, and has a well-defined order. Let us see some lists in action.

```
>>> []  # the empty list
[]
>>> my_list = [5]
>>> my_list
[5]
>>> my_list.append(8)
>>> my_list
[5, 8]
>>> my_list + my_list  # sensible arithmetic operations are defined
[5, 8, 5, 8]
>>> my_list * 3  # such as multiplication by integers
[5, 8, 5, 8, 5, 8]
>>> [5] + [[5]] + [[[5]]]  # lists can contain other lists, and elements may be of any type
[5, [5], [[5]]]
```

Individual elements of lists may be accessed according to their *index* (their position in the list) *via* the indexing operator, `[]`,

```
>>> my_list = [[1, 2], [3, 4], [5, 6]]
>>> my_list[0]
[1, 2]
>>> my_list[1][1]
4
>>> my_list[0::2]
[[1, 2], [5, 6]]
>>> my_list[::-1]
[[5, 6], [3, 4], [1, 2]]
>>> my_list[0] = 12  # assigning to a element changes the list
>>> my_list
[12, [3, 4], [5, 6]]
```

The expression `my_list[a:b:c]` evaluates to a list containing every c^{th} element of `my_list` starting with the a^{th} element proceeding up to, but not including, the b^{th} element. If c is negative, the order is reversed. If a or b is negative, then the index is with respect to the last element, working backwards, *i.e.*, `l[-1]` evaluates to the last element of `l`, `l[-2]` the second to last, etc.

```
>>> numbers = list(range(12))
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> numbers[1:9:2]
[1, 3, 5, 7]
```

The length of a list (as well as any other object with a `__len__` method—a *sized* object) can be interrogated with the `len` function,

```
>>> l = [0, 1, 2]
>>> len(l)
3
>>> l.append(3)
>>> len(l)
4
```

1.3.2 The `for` statement (definite iteration)

Previously we saw the `while` statement, which executes a block of code until some condition is met. The `for` statement is used when we wish the execute a block of code a *fixed number* of times, for example, once for every atom in a molecule. The `for` statement has the form

```
for target_list in expression_list:
    suite
else:
    suite_b
other code
```

Taken from the Python reference,

> The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The `suite` is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), the `suite` in the `else` clause, if present, is executed, and the loop terminates.

Consider the following simple example of taking the arithmetic mean of the elements of a list,

```
1 l = [0, 1, 2, 3, 4]
2 average = 0
3 for x in l:
4     average += x
5 else:
6     average /= len(l)
```

When execution reaches line 3, an iterable is created from the list `l`. The first value in `l`, `0`, is assigned to the name `x`. `x` is then added to average on line 4, which now equals `0`. We have reached the end of the suite, so we return to line 3, where the next value in l, `1`, is assigned to the name `x`. The suite is executed a total of 5 times, once for every element of `l`. Finally, the suite associated with the `else` branch is executed. The `len` function on line 6 takes an iterable as its argument, and returns the number of elements in the iterable (here, 5).

Info: `average += x` and `average /= len(l)` are *augmented assignment statements*. They can be written as the ordinary assigment statements `average = average + x` and `average = average / len(l)`, and achieve the same effect, but are somewhat faster because:

a. they evaluate the target one fewer time (here, the target is `average`);

b. where possible, they directly modify the object to which the target was originally assigned, rather than creating a new object and assigning it to the target.

Using the `len` function, the above `for` loop could be re-written as a `while` loop,

```
l = [0, 1, 2, 3, 4]
average = 0
n = len(n)
i = 0
while i < n:
    average += l[i]
    i += 1
average /= n
```

Of course, the whole exercise could be avoided by using the built-in function `sum`.

1.3.3 Functions of lists

`zip`

The `zip` function allows us to iterate over the elements of two or more lists, pairing elements with the same index.

```
>>> for a, b, c in zip([0, 1, 2], [3, 4, 5], [6, 7, 8]):
...     print(a, b, c)
...
0 3 6
1 4 7
2 5 8
```

On line 36 of *pairpotential.py*, `zip` is used to iterate over the elements of `x1` and `x2`, the coordinates of a pair of distinct particles,

```
36      for x1, x2 in combinations(x, 2):  # for statement
37          r_squared = 0.0
```

Notice that the dimension of the system is not explicitly given; the lists `x1` and `x2` could contain 2, 3, or more float coordinates. This code is designed to work for a system of any dimension. In the three-dimensional case, the suite of the `for` loop starting at line 36 is executed three times, one for each pair of x-, y-, and z-coordinates.

Info: Zippers beware! The iterable returned from by `zip` will yield as many items as there are in the shortest argument passed to it,

```
>>> for a, b, c in zip([0, 1, 2], [3, 4], [6]):
...     print(a, b, c)
...
0 3 6
```

In the above example, the shortest argument, `[6]`, has length one, so the iterable produced yields only one item.

`itertools.combinations`

The `combinations` function in the `itertools` module allows us to iterate over non-identical n-tuples of elements in the same list. `combinations` takes two arguments: a list (or any other *iterable*), and an integer that specifies whether pairs, triples, ..., n-tuples of elements should be returned.

```
>>> for a,b in combinations([0,1,2], 2):
...     print(a,b)
...
0 1
0 2
1 2
>>> for a,b,c in combinations([0,1,2,3], 3):
...     print(a,b,c)
...
0 1 2
0 1 3
0 2 3
1 2 3
```

If tuples including repeated elements are required, the `product` function, also in the `itertools` module, can be used. `product(iterable, repeat=n)` effectively implements an n-level nested loop; *i.e.*,

```
for x, y, z in product(l, repeat=3):
    print(x, y, z)
```

has the same effect as

```
for x in l:
    for y in l:
        for z in l:
            print(x, y, z)
```

The approaches are equally computationally efficient, but the first example is much easier to read. On line 34 of *pairpotential.py*, `combinations` is used to iterate over every pair of non-identical particles,

```
34      energy = 0.0
35
36      for x1, x2 in combinations(x, 2):  # for statement
37          r_squared = 0.0
38          for c1, c2 in zip(x1, x2):  # for statement
39              r_squared += (c1 - c2) * (c1 - c2)
```

If the system contains n particles, the suite of the `for` loop beginning at line 34 is executed $\binom{n}{2}$ times.

1.3.4 Functions and their arguments

What now remains is to explain the function call to `potential` on line 39. From the perspective of the `pair_potential` function, it doesn't matter how many arguments the *callable* `potential` might take, or what their names may be—all that matters is that the arguments are passed to `potential`, and in the correct order (however, the type hint indicates that the function should accept at least one float argument, and return a float).

The arguments are passed to `pair_potential` as a tuple, which is *unpacked* in the function call to `potential` with the `*` operator. The call

```
pair_potential(x=[[0.0,0.0],[0.0,1.0],[0.0,2.0]], potential=lj, potential_args=(1.0, 1.0))
```

results in the call

```
energy += potential(r, 1.0, 1.0)
```

This syntax allows the function `pair_potential` the flexibility to work with more than one kind of *function signature* (here, potential functions with different numbers of arguments).

Info: Like a `list`, a `tuple` is a container type. Unlike lists, tuples are *immutable*—once a tuple object has been created, it cannot be modified. We can use `len` to find the length of a tuple, and `[]` to access the elements of a tuple, but, since they are immutable, assigning to an element raises an exception, and the `append` method is not defined.

```
>>> ()  # empty tuple
()
>>> (1,)  # single-element tuple---notice the terminal comma, ","!
(1,)
>>> (1)  # this is an integer!
1
>>> (1, 2)  # two-element tuple---no terminal comma necessary
(1, 2)
>>> tup = (1, 2, 3)
>>> tup
(1, 2, 3)
>>> len(tup)  # length
3
>>> tup[0]  # indexing
1
>>> tup[0::2]  # slicing
(1, 3)
>>> tup[0] = 4  # assigning---not supported
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tup.append(3)  # appending---not supported
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> tup[0]
```

In some situations, a list is the most appropriate container type; in others, a tuple is preferred. For example, when it is necessary to modify the contents of the container, a list is more useful than a tuple. When it is necessary that the elements of the container be read-only, a list *can* be used, but a tuple is more suited to the problem, as an error is raised when writing to an element is attempted.

Question 3 Prime Numbers

Using your new-found knowledge of container datatypes, write a function that returns a list of its argument's prime factors, e.g.,

```
>>> prime_factors(1)
[]
>>> prime_factors(2)
[2]
>>> prime_factors(60)
[2, 3, 5]
```

Include a documentation string with tests and **make sure your code passes its own tests!**

Optional: write a function `is_prime`, that returns `True` if its argument is prime, and `False` otherwise. Make your function as efficient as possible.

1.4 Scientific programming

By the end of this section, you should be able to understand all of the python code in `simulations.potentials` and `simulations.pair_potential`, and have a vague understanding of what code like *timing_pair_potential.py* is supposed to do.

The original `simulations.pair_potential` function calculates the pairwise-additive energy of a system of particles interacting *via* an unspecified `potential` function. This calculation is done *via* a pair of nested loops: the outer loop to iterate over particle pairs; the inner loop to iterate over particle coordinates,

```
6  def pair_potential(xs: [[float]], potential: Callable[[float, ...], float],
7                     potential_args: tuple = ()) -> float:
34     energy = 0.0
35
36     for x1, x2 in combinations(x, 2):  # for statement
37         r_squared = 0.0
38         for c1, c2 in zip(x1, x2):  # for statement
39             r_squared += (c1 - c2) * (c1 - c2)
40         r = sqrt(r_squared)  # sqrt imported from math module
41         energy += potential(r, *potential_args)
42
43     return energy
```

(docstrings omitted for brevity).

The function `pair_potential_half_vectorised` achieves the same effect, but replaces the inner loop with a *vectorised* calculation using the `numpy` package.

```
8  def pair_potential_half_vectorised(xs: ndarray, potential: Callable[[float, ...], float],
9                                     potential_args: tuple = ()) -> float:
36     energy = 0.0
37
38     for x1, x2 in combinations(xs, 2):  # for statement
39         r = np.linalg.norm(x1-x2)
40         energy += potential(r, *potential_args)
41
42     return energy
```

`pair_potential_vectorised` is a similar function with both loops replaced with vectorised code, the result being significantly faster than our original `pair_potential` implementation.

```
7  def pair_potential_vectorised(xs: ndarray, potential: Callable[[ndarray, ...], ndarray],
8                                potential_args: tuple = ()) -> float:
35     nparticles, ndim = xs.shape
36     left_indices, right_indices = np.triu_indices(nparticles, k=1)
37     rij = xs[left_indices] - xs[right_indices]
38     dij = np.linalg.norm(rij, axis=1)
39
40     return potential(dij, *potential_args).sum()
```

To demonstrate this difference, the script *timing_pair_potential.py* contains functions to compare the speed of the three implementations as a function of particle number, and plots the results using the `matplotlib` package.

1.4.1 NumPy

NumPy, short for **Num**erical **Py**thon, is a python package for scientific programming. It provides a framework for efficiently dealing with large amounts of data, and implements many algorithms from linear

algebra, Fourier transforms, and efficient random number generations, to name just three fields. The key component of the NumPy module is the *n-dimensional array*, or *ndarray*.

The ndarray, like the list, is a container type, but with several important differences. Ndarrays can be initialised with the `array` function, which takes a list as its argument,

```
>>> import numpy as np
>>> a = np.array([0, 1, 2])
>>> type(a)
<class 'numpy.ndarray'>
>>> a
array([0, 1, 2])
```

The list can contain any numeric type, and the list elements can be cast to another type with the `dtype` keyword argument,

```
>>> a = np.array([0.0, 1.0, 2.0])
>>> a
array([ 0.,  1.,  2.])
>>> a = np.array([0, 1, 2], dtype=float)
>>> a
array([ 0.,  1.,  2.])
>>> a = np.array([0, 1, 2], dtype=complex)
>>> a
array([ 0.+0.j,  1.+0.j,  2.+0.j])
```

N-dimensional ndarrays can be initialised by passing a list of lists (of lists…etc) as the argument,

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

The dimensions of the ndarray are stored in the `shape` attribute, and the total number of elements can be accessed *via* the `size` attribute. The shape can be changed *via* the `reshape` method,

```
>>> a = np.array([[[0, 1], [2, 3], [4, 5]]])
>>> a
array([[[0, 1],
        [2, 3],
        [4, 5]]])
>>> a.shape
(1, 3, 2)
>>> a.size
6
>>> b = a.reshape(3, 1, 2)
>>> b
array([[[0, 1]],

       [[2, 3]],

       [[4, 5]]])
>>> b.shape
(3, 1, 2)
>>> c = a.reshape(-1)
>>> c
```

```
array([0, 1, 2, 3, 4, 5])
>>> c.shape
```

Using the `len` function on arrays will return the left-most element of the shape tuple,

```
>>> a = np.array([[[0, 1], [2, 3], [4, 5]]])
>>> len(a) == a.shape[0]
True
```

While lists can contain objects of any type, ndarrays are homogeneous. If the elements of the argument list can be cast to the same dtype, an ndarray with that dtype will be created,

```
>>> a = np.array([1, 2, "string"])  # array of strings!
>>> a
array(['1', '2', 'string'],
      dtype='<U21')
```

Numeric-type ndarrays are contiguous, *i.e.*, have a rectangular shape. If a list of lists of different lengths is passed as the argument, an object-type ndarray is created,

```
>>> a = np.array([[[0,0],[1,1]],[[0,1,2,3],[0,1,2,3]]])
>>> a
array([[[0, 0], [1, 1]],
       [[0, 1, 2, 3], [0, 1, 2, 3]]], dtype=object)
>>> a.shape
(2, 2)
```

If the lists have different depths (*e.g.* a list and a list of lists), an exception is raised,

```
>>> a = np.array([1, [2], [[3]]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: setting an array element with a sequence.
```

Ndarrays filled with zeros or ones can by constructed with the `zeros` and `ones` functions, as well as the `zeros_like` and `ones_like` functions. These functions are useful for initialising arrays. The former take the shape of the array as the first argument, and an optional dtype; the latter take another array whose shape is to be copied, along with an optional dtype,

```
>>> a
array([ True,  True,  True,  True,  True], dtype=bool)
>>> b = np.zeros_like(a, dtype=float)
>>> b
array([ 0.,  0.,  0.,  0.,  0.])
```

ndarrays support the same kind of indexing as lists, as well as `fancy indexing`, which allows a list or another ndarray to be passed as the argument to the indexing operator,

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

Normal indexing returns a *view* of an array; changes to a view of an array are also applied to the array itself,

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

Fancy indexing creates a *copy* of an array; a new array object,

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

Be careful to take notice whether you are working with views or copies.

Tip If an array view is passed to a function, and the function changes the view, the orignal array will also be changed! This is also true of lists, and a source of consternation for beginners and experts alike. If a function changes its arguments rather than creating a new object, it is said to change the argument *in place*. A pure function does not change its arguments; impure functions are not allowed in programs written in purely functional languages. Python is very liberal in this regard—it is up to you to make sure that the user knows whether a function modifies its arguments in place, by writing clear documentation.

Operations

The arithmetic and comparison operators work on an element-wise basis,

```
>>> a = np.array([0, 1, 2, 3, 4, 5])
>>> a * 5
array([ 0,  5, 10, 15, 20, 25])
>>> b = np.array([5, 4, 3, 2, 1, 0])
>>> a * b
array([0, 4, 6, 6, 4, 0])
>>> a ** b
array([0, 1, 8, 9, 4, 1])
>>> a < b
array([ True,  True,  True, False, False, False], dtype=bool)
```

The boolean array that results from a comparison operation can be used as a filter,

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

Since python3.5, matrix multiplication can be acheived with the `@` operator,

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a @ a
array([[ 7, 10],
       [15, 22]])
>>> b = np.array([0, 1, 2])
>>> b @ b  # dot product
5
```

For arithmetic and comparison operators to succeed, the two arrays must have the same shape; otherwise, a `ValueError` is raised. Dummy dimensions can be added by passing `None` or `np.newaxis` to the indexing operator,

```
>>> a = np.array([0, 1, 2])
>>> b = np.array([[0, 1, 2]])
>>> a @ b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (3,) and (1,3) not aligned: 3 (dim 0) != 1 (dim 0)
>>> a[:, np.newaxis] @ b
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
>>> a[:, np.newaxis] @ a[np.newaxis, :]  # outer product
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
>>> a[np.newaxis, :] @ a[:, np.newaxis]  # inner product
array([[5]])
```

Functions

NumPy provides many useful functions for operating on ndarrays. There are NumPy versions of many built-in functions and functions in built-in modules,

```
>>> import numpy as np
>>> a = np.array([0,1,-2])
>>> np.abs(a)
array([0, 1, 2])
>>> np.sum(a)
3
>>> np.cos(a)
array([ 1.        ,  0.54030231, -0.41614684])
```

as well as other convenient definitions,

```
>>> np.std(a)  # standard deviation of a sample
0.816496580927726
>>> np.linalg.norm(a)  # norm of a vector
2.23606797749979
```

When applying reduction operations (`np.sum` , `np.linalg.norm` , etc), an axis can be specified,

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> np.sum(b, axis=0)
array([12, 15, 18, 21])
>>> np.linalg.norm(b, axis=1)
array([ 3.74165739, 11.22497216, 19.13112647])
```

Attempting to apply a function from the `math` module to an ndarray will usually result in a `TypeError` ,

```
>>> from math import cos
>>> cos(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only size-1 arrays can be converted to Python scalars
```

Vectorisation

The NumPy operations and functions we have seen so far are *vectorised*. Let's say we want to calculate the cosine of a thousand numbers in an ndarray. We could iterate over this array in a `for` loop, calculating the cosine of each number in sequence, and storing it somewhere useful. However, since the cosine of the first number doesn't depend on the cosine of the second, third ... or thousandth number, we could, in principle, calculate them all at the same time! That's what happens when we call the vectorised `np.cos` function. Independent cosines are computed simultaneously, with the number of concurrent calculations depending on the hardware. The result, for medium to large arrays, is a significant reduction in compute time. For small arrays, the difference might be unnoticeable, and for very small arrays, the vectorised calculation may in fact be slower.

Now we return to our `pairpotential*` series, to see how vectorisation can help us to write efficient code.

The original `pair_potential` contains a two-level nested loop,

```
34      energy = 0.0
35
36      for x1, x2 in combinations(x, 2):  # for statement
37          r_squared = 0.0
38          for c1, c2 in zip(x1, x2):  # for statement
39              r_squared += (c1 - c2) * (c1 - c2)
40          r = sqrt(r_squared)  # sqrt imported from math module
41          energy += potential(r, *potential_args)
```

Lines 34–39 accumulate the squared distance between two particles in a loop, considering each dimension sequentially, and find the distance as its square root. The function `pair_potential_half_vectorised` in instead contains a call to `np.linalg.norm`, calculating the interparticle distance directly from the particle coordinate arrays,

```
38      for x1, x2 in combinations(xs, 2):  # for statement
39          r = np.linalg.norm(x1-x2)
40          energy += potential(r, *potential_args)
```

The remaining loop can be removed by noticing the following: we want to express the entire distance calculation in a series of vector operations. We seek an expression of the form,

$$r_{ij} = r_j - r_i$$

where the rows of r_i and r_j put together each constitute a unique interparticle vector.

We achieve this through fancy indexing. The indices we want are the row and column indices of the upper triangle of an $N \times N$ matrix, without the diagonal indices (self interactions), where N is the number of particles (the first element of `x.shape`).

The functions `pair_potential_vectorised` obtains these indices using the NumPy function `triu_indices`, which returns two arrays containing the relevant row and column indices,

```
35      nparticles, ndim = xs.shape
36      left_indices, right_indices = np.triu_indices(nparticles, k=1)
37      rij = xs[left_indices] - xs[right_indices]
38      dij = np.linalg.norm(rij, axis=1)
```

We then compute the distances with `np.linalg.norm`, reducing over the last axis.

This description may be somewhat opaque, so let's look at an example. If we have three particles, we need to calculate three unique distances, between particles one and two, one and three, and two and three. So, in r_i we want the rows to be the coordinates of particles (1, 1, 2). Likewise, in r_i we want the rows to be the coordinates of particles (2, 3, 3). The call `triu_indices(3, k=1)` provides exactly that,

```
import numpy as np
>>> coordinates = np.array(["one", "two", "three"])
>>> left_indices, right_indices = np.triu_indices(3, k=1)
>>> left_indices
```

```
array([0, 0, 1])
>>> right_indices
array([1, 2, 2])
>>> coordinates[left_indices]
array(['one', 'one', 'two'],
      dtype='<U5')
>>> coordinates[right_indices]
array(['two', 'three', 'three'],
      dtype='<U5')
```

Truth

Do we need to change `lj_potential` to take advantage of vectorisation? A little.

Since there are no loops in `lj_potential`, and arithmetic and comparison operators are already vectorised, we don't need to worry about the implementation of the energy calculation.

There are two small changes that need to be made. We need to modify our doctests to use arrays,

```
    >>> from numpy import array
    >>> lj_potential_vectorised(array([1.0, 2**(1/6), 2.0]), epsilon=1.0, sigma=1.0)  #doctest: +ELLIPSIS
    array([ 0.        , -1.        , -0.0615...])
```

and to make some changes to our exceptions.

Truth testing is ambiguous for arrays with more than one element. That is to say, an ndarray with more than one element is neither true nor false. As such, the statement `if r < 0.0:` in the original `lj_potential` will raise a `ValueError`. To test if any, or all elements of an array are "true", we can use the `any` and `all` functions. `any(a)` returns `True` if at least one element of `a` is a true value, and `False` otherwise; `all(a)` returns `True` if all elements of `a` are true values, and `False` otherwise.

In our case, we want to check whether *any* element of `r` is less than zero, and whether *any* element of `sigma` is less than or equal to zero, so we use the *np.any* function,

```
    if not isinstance(rs, ndarray):
        raise TypeError(f'rs should be ndarray, not {type(rs).__name__}')
    if np.any(rs <= 0):
        raise ValueError(f'all r must be positive, but min(r) = {rs.min()}')
    if epsilon <= 0 or sigma <= 0:
        raise ValueError(f'both epsilon and sigma must be positive, not ({epsilon}, {sigma})')
```

In addition to `np.any` and `np.all`, the `np.where` function can be used to effect element-wise `if...else`,

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

Our new vectorised version of `lj_potential` is then,

```
import numpy as np
from numpy import ndarray


def lj_potential_vectorised(rs: ndarray, epsilon: float = 1.0, sigma: float = 1.0) -> ndarray:
    """
    compute the Lennard Jones potential at particle separation r,

    V_LJ = 4 epsilon ( (sigma/r)^12 - (sigma/r)^6 )

    >>> from numpy import array
```

```
    >>> lj_potential_vectorised(array([1.0, 2**(1/6), 2.0]), epsilon=1.0, sigma=1.0)  #doctest: +ELLIPSIS
    array([ 0.        , -1.        , -0.0615...])

    >>> lj_potential_vectorised(-1)
    Traceback (most recent call last):
    TypeError: rs should be ndarray, not int

    >>> lj_potential_vectorised(array([1.0, 2**(1/6), -1.0]))
    Traceback (most recent call last):
    ValueError: all r must be positive, but min(r) = -1.0

    >>> lj_potential_vectorised(array([1.0, 2**(1/6), 2.0]), epsilon=-1.0, sigma=1.0)
    Traceback (most recent call last):
    ValueError: both epsilon and sigma must be positive, not (-1.0, 1.0)
    """

    if not isinstance(rs, ndarray):
        raise TypeError(f'rs should be ndarray, not {type(rs).__name__}')
    if np.any(rs <= 0):
        raise ValueError(f'all r must be positive, but min(r) = {rs.min()}')
    if epsilon <= 0 or sigma <= 0:
        raise ValueError(f'both epsilon and sigma must be positive, not ({epsilon}, {sigma})')

    r6 = (sigma / rs) ** 2
    r6 *= r6 * r6
    return 4 * epsilon * r6 * (r6 - 1)
```

1.4.2 Matplotlib

Matplotlib is a python package that provides functions to draw figures. You can get started with Matplotlib very quickly,

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> a = np.linspace(0,1,100)
>>> plt.plot(a, a*a)
>>> plt.show()
```

Here, the function `plt.plot` is given two positional arguments, which are arrays of equal size. This function creates the plot object, but `plt.show` must be called to display the plot. If you type the above commands into the console, you should get a plot of $y = x^2$ for $x \in [0, 1]$.

To have more control over your figures, it is best to instantiate a `figure` object and accompanying `axis` objects, as in *timing_pair_potential.py*,

```
    fig, ax = plt.subplots(1, 1, dpi=160, constrained_layout=True, figsize=plt.figaspect(1 / 2))
    ax2 = plt.twinx(ax)
```

where we have created a figure two twinned axes.

Axis labels and other properties of axes can be set *via* a variety of `.set_*` methods of axes objects, and lines can be labelled by passing a `label` keyword argument to any of the many plotting functions (if you add labels, don't forget to call `ax.legend`. We move all of these details into a separate function, `modify_plot`,

```
def modify_plot(fig, ax, ax2, nparticles):
    ax.set_xlabel('number of particles')
    ax.set_ylabel('time / seconds')
    ax.set_xscale('log')
```

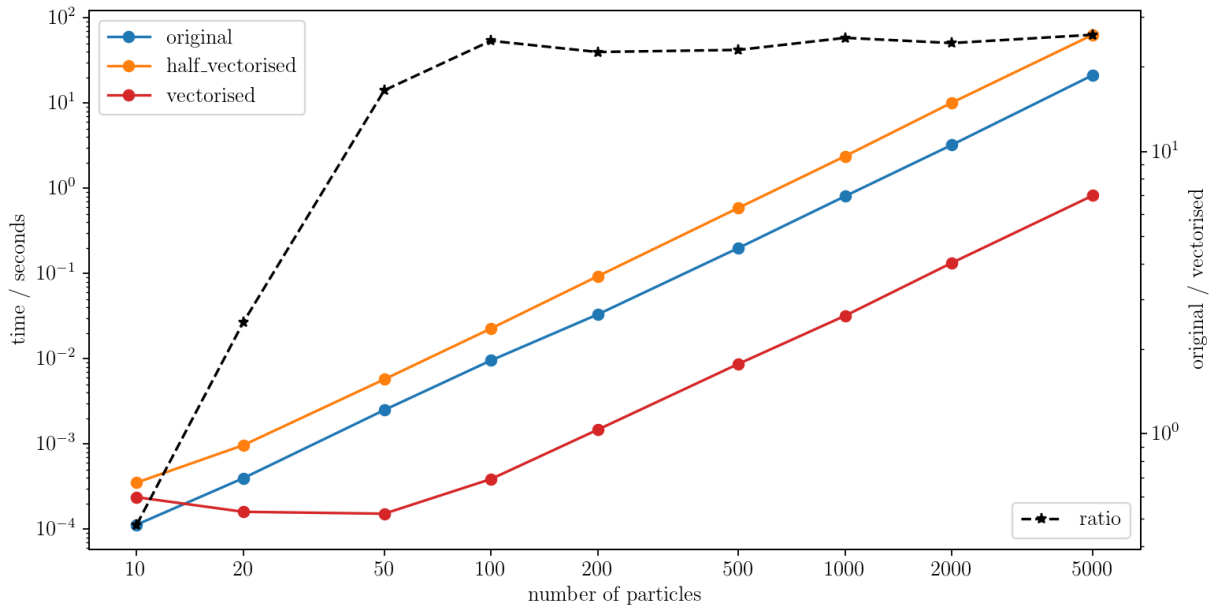

Figure 1.1: Upper panel: A comparison of the execution times of three implementations of the pair potential function mentioned in the text. Lower panel: relative execution time of the two loop implementation compared to the vectorised implementation.

```
36      ax.set_yscale('log')
37      ax.set_xticks(nparticles)
38      ax.set_xticks([], minor=True)
39      ax.set_xticklabels(nparticles)
40      ax.legend()
41
42      ax2.set_ylabel('original / vectorised')
43      ax2.set_yscale('log')
44      ax2.legend(loc=4)
45      return ax
```

Different line styles can be included in call to the plot function (see the Matplotlib documentation for details).

To save a figure rather than plot it to the screen, the `plt.save_fig` function is used, with the file name as its first argument.

Running *timing_pair_potential.py* will compare the execution speed of our three pair potential functions, `pair_potential` , `pair_potential_halfvectorised` , and `pair_potential_vectorised` for different numbers of particles, and save a figure to *timing_pair_potential.png*.

Give it a try! My result is shown figure 1.1, and indicates that, on my desktop, the vectorised code we saw in this section executes about 25 times faster than that from the last section for large particle number.

Info: The `__main__` block of *timing_pair_potential.py* is executed when *timing_pair_potential.py* is the main program; if the main program imports something from *timing_pair_potential.py*, the `__main__` block will be ignored.

Question 4 Molecular Dynamics Prep.

1. **Forces.** Write a function that calculates the per-particle forces in a system of Lennard-Jones particles. First write a naive loop to make sure you get the right result, then implement a vectorised version.

Chapter 2

Numbers and Precision

2.1 Decimal Numbers

There are many numbers—uncountably many, in fact. If a language were to assign to each number a unique word, that language would similarly have infinitely many words. In China, a senior high-school student is expected to learn ≈ 6600 characters, yet most can discuss numbers beyond 10000.

Most advanced civilisations have relied upon number systems that represent numbers as some expansion of powers of some base. For example, in Ancient Egypt, powers of ten from one to one million each had their own symbol (hieroglyph), with other numbers expressed by writing the appropriate number of copies of those basic symbols (table 2.1). Such a system has the number ten as its base, and is referred to as a decimal number system.

In the prevailing positional decimal number system, integers are expressed as decimal numerals, ordered lists of numerals 0–9, encoding a decimal expansion,

$$a_n a_{n-1} \dots a_1 a_0 = \sum_{i=0}^{n} a_i \times 10^i. \tag{2.1}$$

Real numbers may be expressed as two such numerals interposed by a decimal point, .,

$$a_n a_{n-1} \dots a_1 a_0 . b_1 b_2 \dots b_{m-1} b_m = \sum_{i=0}^{n} a_i \times 10^i + \sum_{j=1}^{m} b_j \times 10^{-j} \tag{2.2}$$

Typically, n, m are chosen such that $a_n \neq 0$, $b_m \neq 0$. The decimal expansion of irrational numbers, such as π, is infinite. Such numbers are usually quoted to an appropriate number of significant digits, *e.g.*, $\pi = 3.1415$ (5 s.f.). Additionally, some rational numbers, such as $1/7$, have infinitely repeating decimal expansions; to handle such cases, a variety of notations are used, such as writing a line over the repeating sequence,

$$\frac{1}{70} = 0.0\overline{142857}. \tag{2.3}$$

2.2 Binary Numbers

The popularity of the base-ten number system may have something to do with the fact that humans each possess ten fingers (digits). Bases eight, twelve, twenty, and sixty have also seen widespread use. Modern

Arabic Numeral	1	10	100	1000	10000	100000	1000000
Egyptian Hieroglyph							
Description	single stroke	cattle hobble	coil of rope	lotus	bent finger	tadpole	Heh

Table 2.1: The Ancient Egyptian decimal system. The number 2021 is written as  .

computing is almost entirely based on the base-two, or binary number system, owing to the ease with which this system can be implemented in circuits using logic gates.

In a positional binary system, each digit represents a different power of two. For instance,

$$1010_2 = 2^3 + 2^1 = 10, \tag{2.4}$$

where the subscript of the left-hand-side indicates the base. Binary fractions are similar have a similar interpretation to their decimal counterparts,

$$10.101_2 = 2^1 + 2^{-1} + 2^{-3} = 2 + \frac{1}{2} + \frac{1}{8} = 2.625 \tag{2.5}$$

In Python, a binary integer literal is the flag `0b` followed by a sequence of zeros and ones. The binary representation of a decimal number can be obtained (as a string) *via* the `bin` function,

```
>>> 0b1010
10
>>> bin(10)
'0b1010'
```

Info: The equivalent flags and functions for octal (base 8) and hexadecimal (base 16) are `0o, oct` and `0x, hex`.

Any finite binary fraction can be expressed as a finite decimal fraction,

$$\sum_i b_i \times 2^{-i} = \sum_i b_i \times \frac{1}{2^{-i}} \frac{10^i}{10^i} = \sum_i 5^i b_i \times \frac{1}{10^{-i}}, \tag{2.6}$$

but the converse is not true—in base two, the decimal number 0.1 is an infinitely repeating fraction,

$$0.1 = 0.0\overline{0011}_2 \tag{2.7}$$

Although 0.1 and numbers like it cannot be represented exactly as binary fractions, it is still possible to represent them exactly as tuples of integers (a numerator and a denominator).

2.3 Integers

The values representable by an integer type are determined by (1) the size of that type, *i.e.*, the amount of memory allocated to it, and (2) whether or not the type is signed. For instance, a 32-bit unsigned integer i can take values between $0 \leq i < 2^{32}$; each of the 32 bits represents a term in the binary expansion of the number. A 32-bit signed integer j can take values between $-2^{31} \leq j < 2^{31}$; here, 31 bits each represent a term in the expansion, and one bit encodes the sign.

In Python3, the default integer type is unbounded, *i.e.*, there is no limit on the numerical size of the number (hardware limitations still apply). However, numpy integers are bounded. If you try to exceed the bounds of a bounded integer type, you will discover that integer arithmetic is actually implemented as modular arithmetic,

```
>>> from numpy import full, int32, int64
>>> arr = full(1, 2**31-1, dtype=int32)
>>> arr
array([2147483647], dtype=int32)
>>> arr+1
array([-2147483648], dtype=int32)
```

Exceeding the bound of a type is handled be wrapping around to the other bounding value. This situation is known as *integer overflow*.

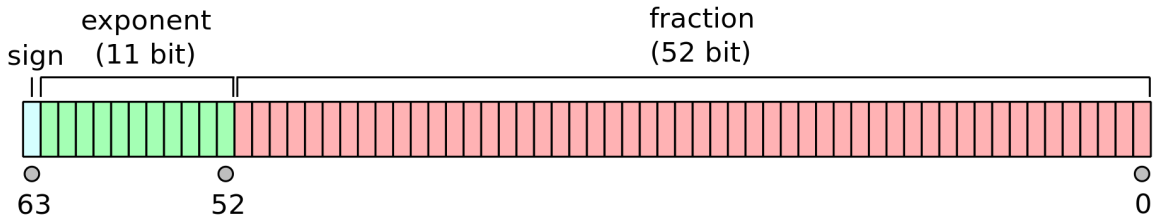

Figure 2.1: The anatomy of a IEEE-754 double-precision float: 1 bit is reserved for the sign, 11 bits for the exponent, and 52 bits for the fraction. If the exponent is non-zero, the number is normal.
Figure taken from here.

2.4 Floating Point Numbers

Numbers other than the integers can be represented using floating-point types. It is often convenient to write such numbers in scientific notation, *i.e.*, a product of a sign, a fractional part (the mantissa), and a base (radix) taken to the power of some exponent,

$$(\text{sign})\,\text{fraction} \times \text{base}^{\text{exponent}} \tag{2.8}$$

The number $-1/8$ is thus represented as -1.25×10^{-1} in decimal and $-1_2 \times 2^{-3}$ in binary.

The most widely-used are the single- and double-precision floating-point numbers (singles and doubles) specified in the IEEE-754 standard. Singles require 32 bits per number: the largest bit encodes the sign, the next 8 bits encode the exponent, and the last 23 bits encode the fraction. Doubles require 64 bits: 1 for the sign, 11 for the exponent, and 52 for the fraction (figure 2.1). Because they involve manipulating less data, operations between singles are faster than between doubles (at least twice as fast, sometimes up to 32 times faster), but computations with doubles are more precise (they can be specified to a larger number of significant figures); in computational physics, the need for higher precision usually wins out, and so doubles are preferred over singles.

2.4.1 Fractional Part

The fractional part, b, actually encodes a 53-bit number, with a largest bit equal to 1,

$$(1.b_{52}b_{51}\ldots b_2b_1)_2 \tag{2.9}$$

Doubles therefore have 53-bit precision in binary, corresponding to 15–17 significant decimal digits. The maximum relative rounding error when rounding a number to the nearest representable one (known as the machine epsilon) is therefore $\epsilon = 2^{`53} \approx 1.11 \times 10^{-16}$.

2.4.2 Exponent

The range of the exponent string, e, is 0–2047. There are several cases to consider.

- $\mathbf{0 < e < 2047}$ the exponent is given by $e - 1023$, *i.e.*, allowable exponents are integers on the range -1022–1023. In this case, the number encoded is *normal*: it has full precision. The rule for decoding the number is the expression,
$$(-1)^s \times 2^{e-1023} \times (1.b)_2 \tag{2.10}$$
where s is the sign bit. The range of the positive normal numbers is thus 2^{-1022} to 2^{1023}, approximately $2.2250738585072014 \times 10^{-308}$ to $1.7976931348623157 \times 10^{308}$ in decimal.

- $\mathbf{e = 0; b \neq 0}$ these are the **denormal** numbers. Denormal numbers fill the underflow gap around zero, ensuring that the difference between two distinct normal numbers is always non-zero. Denormal numbers are defined to less than 53 bits of precision. The expression to decode the string becomes,
$$(-1)^s \times 2^{-1022} \times (0.b)_2 \tag{2.11}$$

- $\mathbf{e = 0; b = 0}$ the number evaluates to ± 0

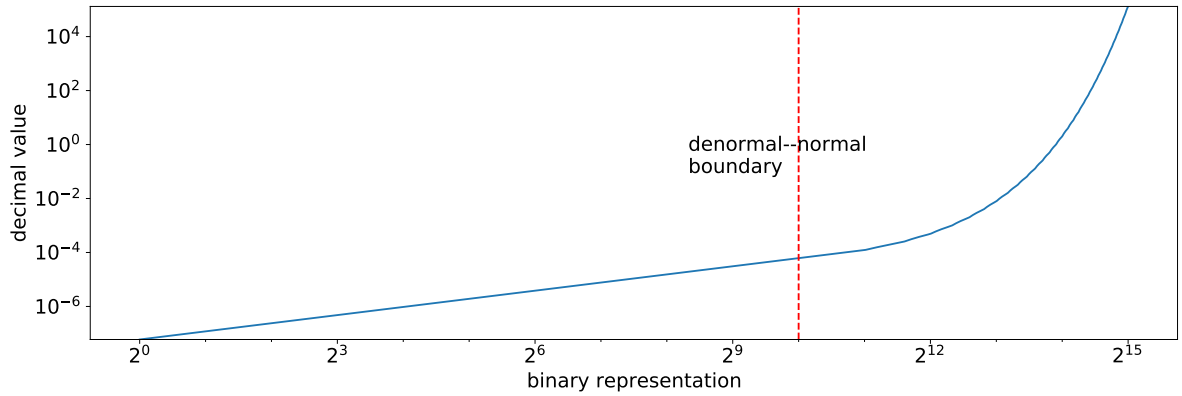

Figure 2.2: Distribution of positive values of 16-bit floats. Subnormal values are linearly spaced, whereas normal values are logarithmically spaced.

- $\mathbf{e} = \mathbf{2047}; \mathbf{b} = \mathbf{0}$ the number evaluates to $\pm\infty$
- $\mathbf{e} = \mathbf{2047}; \mathbf{b} \neq \mathbf{0}$ the number evaluates to NaN (not a number)

Per these definitions, normal floating-point numbers provide a logarithmically-spaced coverage of the number line, whereas the denormal numbers are linearly-spaced. Figure 2.2 illustrates this disparity for positive half-precision floats (16-bits, 1 for the sign, 5 for the exponent, and 10 for the fraction).

2.5 Representation Error and Truncation

As we mentioned in section 2.2, the decimal fraction 0.1 has no finite binary expansion, and as such cannot be represented exactly, and must be truncated in order to be stored; this is a source of representation error. Instead, an input value like 0.1 is converted to the closest binary fraction that fits into the target type. If the target is a double, that means finding the 53-bit integer J and exponent N such that $1/10 \approx J/2^N$, or, rearranging, $J \approx 2^N/10$. Noticing that $8 < 10 < 16$,

$$2^{52} = 2^{N-4} < 2^N/10 < 2^{N-3} = 2^{53} \tag{2.12}$$

so $N = 56$ is the only value for N leaving J with exactly 53 bits. Rounding $2^{56}/10$ to the nearest integer gives

$$J = 7205759403792794 \tag{2.13}$$

As such, best representation of 0.1 in double precision is given by the fraction

$$\frac{7205759403792794}{2^{56}}. \tag{2.14}$$

In Python, we can find the fractional representation of any float using the `to_integer_ratio` method of the `float` type, and the exact decimal value of the stored binary fraction using the `from_float` class method of the `Decimal` class,

```
>>> from decimal import Decimal
>>> x = 0.1
>>> x.as_integer_ratio()
(3602879701896397, 36028797018963968)
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Finally, encoded as a double precision value,

$$0.1 \mapsto 0\,01111111011\,1001100110011001100110011001100110011001100110011010_2. \tag{2.15}$$

2.5.1 $0.1 + 0.1 + 0.1 = 0.3$?

A stark example of representation error is found in the simple statement:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Interrogating the exact decimal expansions of the LHS and RHS, we find that there is a more accurate representation of the number 0.3 than is obtained by the sum $0.1 + 0.1 + 0.1$,

```
>>> Decimal.from_float(0.1+0.1+0.1)
Decimal('0.3000000000000000444089209850062616169452667236328125')
>>> Decimal.from_float(0.3)
Decimal('0.2999999999999999888977697537484345957636833190917968875')
```

Errors like this accumulate over the course of a calculation. For example, to find the mean of the values in a list, you might iterate over the values in the list, adding each to a running total, before finally dividing by the number of values,

```
1  def naive_mean(iterable):
2      total = 0
3      for value in iterable:
4          total += value
5      return total / len(iterable)
```

Applying this to a list of just over a million copies of the float 0.1, we obtain a less than satisfactory result,

```
>>> from sums import naive_mean
>>> values = 2 ** 20 * [0.1]
>>> m = naive_mean(values)
>>> m
0.10000000000154079
>>> abs(0.1-m)
1.5407813913626e-12
```

The error in the mean is four orders of magnitude greater than the machine epsilon. This occurs because when two floats of differing orders of magnitude are added together, some of the bits of the smaller number have to be discarded in order for the sum to fit into the data type. Since 0.1 has only an approximate value as a binary fraction, half of the smallest significant bits are non-zero, and the smallest of them will be lost when adding to the running total.

The problem of lost significance can be changing the order of summation. For example, instead of keeping one running total, we could add together pairs of adjacent elements in the list, producing a new list of half the size. We repeat the procedure on the new list until the new list has only one element, and then divide this element by the length of the original list.

```
8  def clever_sum(iterable):
9      if len(iterable) == 1:
10         return iterable[0]
11     elif (len(iterable) % 2) == 1:
12         return iterable[0] + clever_sum(iterable[1:])
13     else:
14         return clever_sum([a + b for (a, b) in zip(iterable[0::2], iterable[1::2])])
15
16
17 def clever_mean(iterable):
18     return clever_sum(iterable)/len(iterable)
```

This recursive algorithm produces much better results than the naive implementation,

```
>>> from sums import clever_mean
>>> m = clever_mean(values)
>>> m
0.1
>>> abs(0.1-m)
0.0
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

but probably has terrible performance characteristics. The `numpy.mean` method is a better alternative to the builtin sum function, producing an error close to the machine epsilon,

```
>>> import numpy as np
>>> m = np.mean(values)
>>> abs(0.1-m)
2.220446049250313e-16
```

2.6 Catastrophic Cancellation

Catastrophic cancellation is the phenomenon that subtracting good approximations to two nearby numbers may yield a very bad approximation to the difference of the original numbers. Catastrophic cancellation can occur whenever approximations are made, such as with the approximate representation of the number line provided by floating point numbers. Take as a simple example the discriminant of the quadratic equation,

$$ax^2 + bx + c = 0 \to x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.16}$$

If $|4ac/b^2| < \epsilon \approx 10^{-16}$, the one of the roots $x_{1,2}$ will suffer from catastrophic cancellation as its numerator will evaluate to 0. Any further computation that relies on the reciprocal the root will cause an exception, but worse would be for the cancellation to go unnoticed and propagate unhindered.

As we re-wrote the algorithm to compute the sum in the previous section, the solution here is to rewrite one of the roots as a function of the other. Noticing that the product of the roots is always $x_1 x_2 = c/a$, we can evaluate the non-pathological root, say, x_1, and then compute the other as $x_2 = c/ax_1$. The roots are then $x_1 = -b/a, x_2 = -c/b$.

Question 5 Evaluating Functions

Rewrite the following expressions so that they can be safely evaluated in the given limit.

For small h,

$$\sin(x+h) - \sin(x)\,; \tag{2.17}$$

for small x,

$$\frac{1-\cos x}{\sin x}; \tag{2.18}$$

for large N,

$$\int_N^{N+1} \frac{\mathrm{d}x}{1+x^2}; \tag{2.19}$$

for small x,

$$\exp x - 1; \tag{2.20}$$

for large N,

$$\int_N^{N+1} \mathrm{d}x \log(1+x)\,. \tag{2.21}$$

2.7 Condition Number

The condition number, $\kappa(f, x)$, of a function $f(x)$ quantifies the maximum relative rate of change of a function with respect to changes in its arguments. Functions with a low condition number are said to be well-conditioned, and those with a high condition number, ill-conditioned. The outputs of well-conditioned functions are less sensitive to small changes to their inputs arising from, *e.g.*, measurement error or approximation errors. Ill-conditioned functions, by contrast, may have outputs that vary strongly on small changes to their inputs, inducing errors in the output disproportionate to those in the input.

Mathematically, the condition number of a function $f(x)$ is given by,

$$\kappa(f, x) = \lim_{\epsilon \to 0} \sup_{\|\delta x\| \leq \epsilon} \frac{\|\delta f(x)\|}{\|f(x)\|} \Big/ \frac{\|\delta x\|}{\|x\|}. \tag{2.22}$$

For differentiable functions of one variable, this expression simplifies to,

$$\kappa(f, x) = \left| \frac{x f'(x)}{f(x)} \right|, \tag{2.23}$$

which the keen-eyed will recognise as the logarithmic derivative of f divided by the logarithmic derivative of x,

$$\left| \kappa(f, x) = \frac{\mathrm{d}}{\mathrm{d}x} \log f \Big/ \frac{\mathrm{d}}{\mathrm{d}x} \log x \right|. \tag{2.24}$$

The logarithm of the condition number gives an approximate indication of how many digits of accuracy will be lost in addition to those lost through imprecision in approximation.

For functions of many variables we can write,

$$\frac{\|J(\mathbf{x})\|}{\|f(\mathbf{x})\| / \|\mathbf{x}\|}, \tag{2.25}$$

where $J(\mathbf{x})$ denotes the Jacobian matrix of partial derivatives of f at $\mathbf{x}$. Notice that this expresses the relative rate of change of a function with respect to the relative rate of change of $\mathbf{x}$, **the norm of a vector containing its arguments**, not just the relative rate of change of an individual argument.

Info: The ℓ_p vector norm $(p \geq 1)$ of a vector $\mathbf{x} = (x_1, \ldots, x_n)$ is

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}. \tag{2.26}$$

The 2-norm is the familiar Euclidean norm; the set of n-dimensional vectors with the same 2-norm covers the surface of a n-1-dimensional hypersphere; the set of vectors with the same 1-norm, called the taxicab or Manhattan norm, forms the surface of a cross polytope (*e.g.*, in 3 dimensions, the surface of the octahedron).

As p approaches ∞, l_p approaches l_∞,called the infinity, or maximum norm,

$$\|x\|_\infty = \max(|x_1|, \ldots, |x_n|). \tag{2.27}$$

The set of vectors with ∞-norm equal to some constant, c, covers the surface of a hypercube with side length $2c$.

2.7.1 Elementary Functions

We will now apply these results to some elementary functions. In the case of scalar multiplication, $f(x) = ax$, the condition number is,

$$\begin{aligned} \kappa(f, x) &= \left| \frac{x \cdot f'(x)}{f(x)} \right| \\ &= \frac{x \cdot a}{ax} \\ &= 1, \end{aligned} \tag{2.28}$$

norm	ℓ_1	ℓ_2	ℓ_∞
$\kappa(+,x,y)$	$2\frac{\lvert x\rvert+\lvert y\rvert}{\lvert x+y\rvert}$	$\frac{\sqrt{2(x^2+y^2)}}{\lvert x+y\rvert}$	$\max\left(\left\lvert\frac{x}{x+y}\right\rvert,\left\lvert\frac{x}{x+y}\right\rvert\right)$
$\kappa(\times,x,y)$	$\lvert x/y\rvert+\lvert y/x\rvert+2$	$\lvert x/y\rvert+\lvert y/x\rvert$	$\max(\lvert x/y\rvert,\lvert y/x\rvert)$

Table 2.2: Condition number of addition and multiplication for three choices of vector norm.

which suggests that scalar multiplication is always well-conditioned, so the accuracy of the output $f(x)$ will be the same as the accuracy in the input, x.

Multiplication of two variables, $f(x,y)=xy$, has distinctly different characteristics,

$$\begin{aligned}\kappa(f,x,y) &= \frac{\left\|\left(\frac{\partial f}{\partial x},\frac{\partial f}{\partial y}\right)\right\|}{\|f(x,y)\|/\|(x,y)\|}\\ &= \frac{\|(x,y)\|^2}{|(x\cdot y)|}\end{aligned} \tag{2.29}$$

In general, the condition number will differ for different choices of norm. Table 2.2 shows the condition number evaluated using the ℓ_p-norm for $p=1,2,\infty$. For all choices of norm, multiplication of two numbers x,y is well-conditioned (κ is close to 1) when the numbers are similar in magnitude, but becomes ill-conditioned when either $x\gg y$ or $y\gg x$.

It is important to note that the condition number does not tell us what the error in f will be; it simply gives an upper bound on the ratio of the relative error in f and the relative error in $\mathbf{x}$. In the case of multiplying two numbers: take $\mathbf{x}=(a,b)$ and $\delta\mathbf{x}=(\delta a,\delta b)$. Let . Consider the case $(a,b)=(1,10000)\,;\|\delta x\|=1$. The error in x could be mostly in a, mostly in b, or spread between the two.

If the error is entirely in b, then the error in the product is 1; *i.e.*, the same as in the error in the input. If, however, the error is entirely in a, the error in the product is 10000, four orders of magnitude greater than the error in the input! This magnification of error is just below the condition number of f at $\mathbf{x}$.

Figure 2.3 shows histograms of ratios of relative errors for a range of a,b, for random samples of $\delta\mathbf{x}$ with $\|\delta x\|=1$. The histograms are bounded from the right by κ, and are strongly skewed toward κ. As expected, the error in the output is smallest most often when $a=b$.

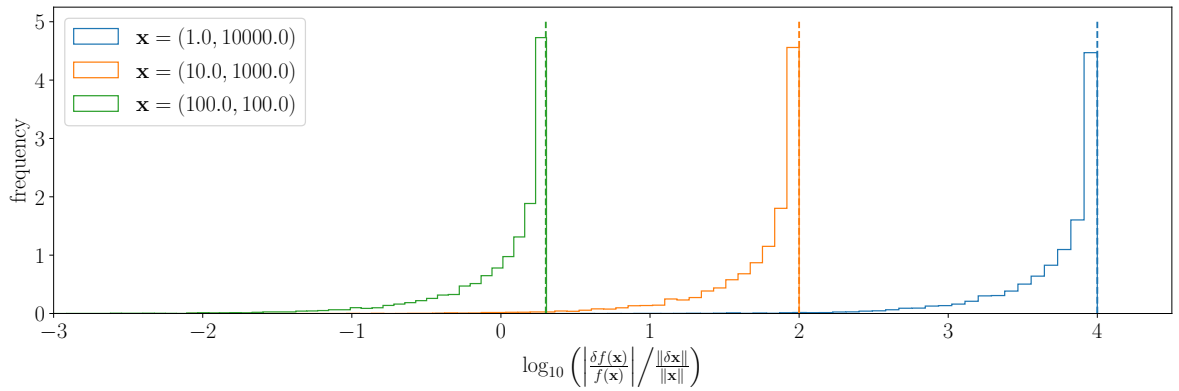

Figure 2.3: Histograms of the logarithm of the ratio of the relative error in $a\cdot b$ and the relative error in $\mathbf{x}=(a,b)$ for a sample of 10,000 error vectors, $\delta\mathbf{x}$, with elements sampled from the standard normal distribution. When $a=b$, the ratio is smallest.

For scalar addition, $f(x) = a + x$, the condition number is,

$$\begin{aligned} \kappa(f, x) &= \left| \frac{x \cdot f'(x)}{f(x)} \right| \\ &= \left| \frac{x}{a + x} \right|. \end{aligned} \tag{2.30}$$

Addition is well-conditioned except near the point $a + x = 0$, *i.e.*, $x = -a$; at this point, the true value of f is exactly 0, and any deviation of x from its true value produces a totally inaccurate result. Addition of two variables, $f(x, y) = x + y$ is also ill-conditioned around $x = -y$; expressions for κ are listed in 2.2.

Referring back to our example of the discriminant of the quadratic equation,

$$ax^2 + bx + c = 0 \rightarrow x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2.16}$$

Fixing the value $a = 1/4$, the expression simplifies to,

$$f(b, c) = 2\left[-b \pm \sqrt{b^2 - c}\right] \tag{2.31}$$

Taking derivatives and plugging in to 2.24, we find,

$$\kappa(f, b, c) = \frac{\sqrt{f^2 + 1}}{f} \cdot \sqrt{\frac{b^2 + c^2}{b^2 - c}}, \tag{2.32}$$

indicating that the discriminant is ill-conditioned not only when $c \ll b^2$, but also when $c \approx b^2$. As c goes to zero, one of the solutions comes close to zero, and either side of $c = b^2$, the number of real solutions changes; both cases constitute catastrophic cancellation resulting from ill-conditioned addition.

Although the condition number of a monomial x^p is $|p|$, that of a polynomial depends on the coefficients, it is difficult to find the roots of ill-conditioned polynomials. Data fitted to an exponential function, $\exp x$, may be pleasing to the eye, but one should be careful to note that its condition number grows as $|x|$, *i.e.*, the number of decimal digits lost grows as $\log(|x|)$, becoming large for both large, negative, and large, positive x.

Chapter 3

Numerical Calculus

Many physical problems can be modelled as systems of differential equations. In some cases, these equations admit to an analytical solution, allowing us to learn no end of interesting information about the problem at hand. Unfortunately, such models are usually very approximate, or limited in their scope, applicable to, or exactly solvable in, a small part of the problem space.

Consider a two-body gravitational problem, such as the Earth orbiting the sun, or the moon orbiting the Earth. By cleverly choosing the coordinate system and applying Newtonian dynamics (see figure 3.1), the time evolution of the positions and velocities of the two bodies can be determined exactly. If we now add a third body, we are stuck—the three-body gravitational problem, excepting special cases, admits of no analytical solution. Similarly, using quantum mechanics, the hydrogen atom (one proton, one electron) was "solved," analytically, long ago; a similar solution for the helium atom (two protons, two electrons) remains wanting.

What are we to do? At this juncture we are left with two options: solve an approximate model, completely; or solve a complete model, approximately. Into a simpler model, that admits to analysis, we can have much deeper insight, but those insights won't necessarily allow us to predict the behaviour of real things. At the other extreme is a "black-box" method, which for some input produces precise and accurate output, but offers little insight about how the two are related. Most of the time, we find ourselves between the two extremes, but the ideas and techniques in this course lean toward the latter approach.

Consider once more the three-body gravitational problem. We don't have an analytical solution, but Newton's equations tell us something about how the system evolves in time,

$$\mathbf{v_i} = \frac{\mathrm{d}\mathbf{r_i}}{\mathrm{d}t} \tag{3.1}$$

$$m_i \mathbf{a_i} = \frac{\mathrm{d}^2\mathbf{r_i}}{\mathrm{d}t^2} = \frac{\partial V}{\partial r_i} \tag{3.2}$$

where $\mathbf{r}_i, \mathbf{v}_i, \mathbf{a}_i$ are the position, velocity, and acceleration vectors of body i, and V is the potential. Given some initial conditions, we could imagine propagating the system through time in small steps, at each step updating the acceleration by computing the force, updating velocity from the acceleration, and then updating the position from the velocity—this is the basis of many numerical integration schemes, and relies on the techniques of numerical calculus.

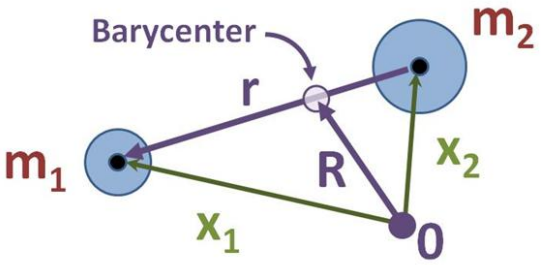

Figure 3.1: Jacobi coordinates for the two-body problem.

3.1 Approximation of a function

3.1.1 Taylor Series

If you are not already familiar with the Taylor series, now is the time to commit it to memory, as they allow us to approximate functions and their derivatives cheaply and robustly.

The Taylor expansion of continuous and differentiable function of single variable, $f(x)$, around the point $x+h$, is given by,

$$f(x+h) = f(h) + xf'(h) + \frac{1}{2}x^2 f''(h) + \frac{1}{6}x^3 f'''(h) + \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!} f^{(n)}(h) \tag{3.3}$$

The special case of $h = 0$ is known as a Maclaurin series. For a function of many variables, $f(\mathbf{x})$, the equivalent is,

$$f(\mathbf{x}+\mathbf{h}) = \sum_{n=0}^{\infty} \frac{(\mathbf{x}\cdot\nabla)^n}{n!} f(\mathbf{h}) \tag{3.4}$$

In practice, we terminate the series at finite order,

$$f(x+h) \approx \sum_{n=0}^{N} \frac{h^n}{n!} f^{(n)}(x) + \mathcal{O}\left(h^N\right) \tag{3.5}$$

where $\mathcal{O}\left(h^N\right)$ is the error term, and means that the absolute value of the error is at most some constant times $\left|h^N\right|$ when h is close enough to zero.

Examples

In the following examples, $h = 0$, and $|x| < 1$ is the necessary and sufficient condition for convergence of the series.

The most straightforward example is that of the exponential functions. To determine its Taylor series expansion around zero, we begin by computing its derivatives,

$$f^{(n)}(0) = e^0 = 1 \quad \forall n \in \mathbb{Z}^+, \tag{3.6}$$

and then substitute these expressions into equation 3.3,

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots. \tag{3.7}$$

The sine function is a little bit more work,

$$\begin{aligned} f^{(4n)}(0) &= +\sin 0 = 0 \\ f^{(4n+1)}(0) &= +\cos 0 = 1 \\ f^{(4n+2)}(0) &= -\sin 0 = 0 \\ f^{(4n+3)}(0) &= -\cos 0 = -1 \quad \forall n \in \mathbb{Z}^+ \end{aligned} \tag{3.8}$$

which leads to,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \tag{3.9}$$

Similarly, the cosine function admits to the expansion,

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots \tag{3.10}$$

The famous equality $e^{i\theta} = \cos\theta + i\sin\theta$ is easily proven by substituting $x \to i\theta$ into these expansions.

Another useful example is the natural logarithm,

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots \tag{3.11}$$

Question 6 Series in a series

Consider the function

$$f(x) = \ln(\cos x), x \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right). \tag{3.12}$$

Using only the Taylor series listed above and elementary algebra, compute the 7th degree Maclaurin polynomial of f (*i.e.*, a polynomial with an error term of order x^8).

3.2 Differentiation

The definition of the first derivative of function of one variable, $f(x)$, at a point x_i, is the limit,

$$f'(x_i) = \lim_{\delta x \to 0} \frac{f(x_i + \delta x) - f(x_i)}{\delta x} \tag{3.13}$$

This limit is well-studied for the elementary functions, and through the application of the product and chain rules, can be determined for functions composed thereof.

If, however, the function is not known in closed form, or we wish to compute derivatives of experimental data whose underlying functions are not known, we cannot rely on analysis to determine the derivative. In this scenario, we must turn to numerical methods. In fact, we can approximate the derivative of a function using Taylor series.

Consider again the Taylor series of a function $f(x)$ evaluated at a point $x + h$, expanded this time around x,

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \dots \tag{3.14}$$

This sequence always converges for $h < 1$. Truncating at second order, we obtain the expression

$$f(x+h) = f(x) + hf'(x) + \mathcal{O}\left(h^2\right) \tag{3.15}$$

Rearranging for $f'(x)$,

$$\begin{aligned} f'(x) &= \frac{f(x+h) - f(x) + \mathcal{O}\left(h^2\right)}{h} \\ &= \frac{f(x+h) - f(x)}{h} + \mathcal{O}\left(h^1\right) \end{aligned} \tag{3.16}$$

This last expression is the *two-point formula* for the first derivative, called so because the function f must be evaluated at two points, $f(x)$ $f(x+h)$. It is also known as the *first forward difference*, since we look at the point x and some point to the right of x, $x + h$.

The last term in equation 3.16 is extremely important: it tells us that the error in our approximation is proportional to h. As h gets closer to zero, the error decreases and the approximation improves.

We can obtain an approximation with better error properties by expanding around both $x + h$ and $x - h$ and taking the difference;

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \mathcal{O}\left(h^3\right) \\ f(x-h) &= f(x) - hf'(x) + \frac{1}{2}h^2 f''(x) + \mathcal{O}\left(h^3\right) \end{aligned} \tag{3.17}$$

The zeroth and second order terms cancel, leaving us with

$$f(x+h) - f(x-h) = 2hf'(x) + \mathcal{O}\left(h^3\right) \tag{3.18}$$

which rearranges to

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}\left(h^2\right), \tag{3.19}$$

wherein the leading error term now scales as h^2. This approximation is known as the *three-point formula*, or *first central differences*, for obvious reasons.

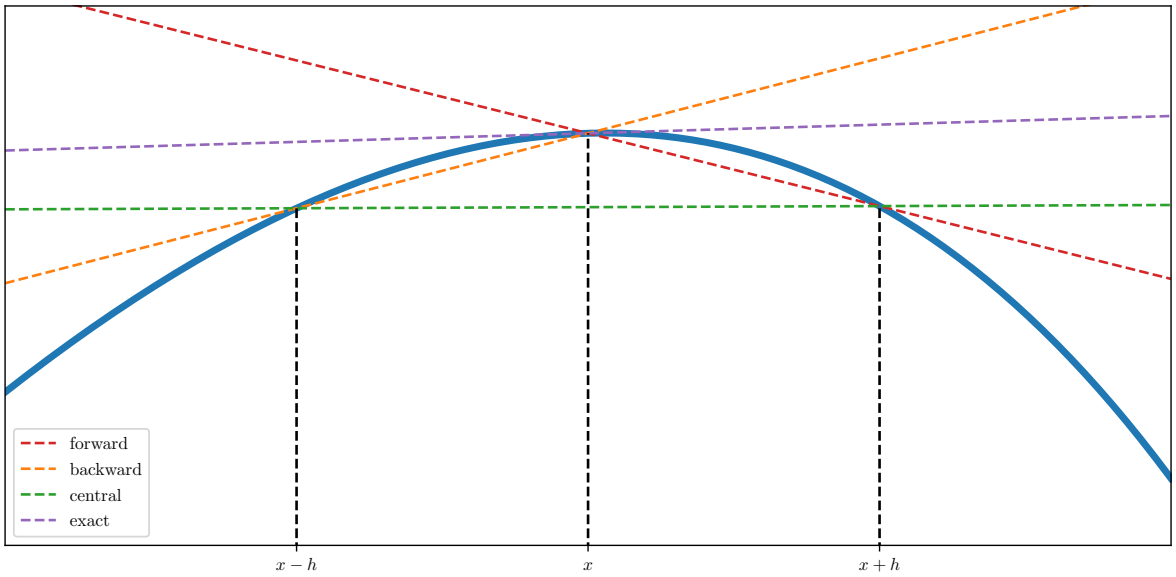

Figure 3.2: Comparison of the first forward, backward, and central differences methods for $f(x) = x - x^2 + x^3 - x^4$ centred at the point $x = 0.6$ with step size $h = 0.1$.

Question 7 Convergence properties of the difference equations

Consider the function $f(x) = xe^x$ at $x = 2$. Compute the first-forward, first-backward, and first-central difference approximations to the derivative of f for different values of h. Plot the error in the approximations as a function of h, and discuss your findings.

Following the idea in equation 3.18, we can use combinations of $f(x \pm nh)$ to cancel still higher order terms, leading to expressions with even tighter error bounds. Using four points we can cancel the third order derivatives in the Taylor expansion, leading to the five-point formula,

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + \mathcal{O}\left(h^4\right) \tag{3.20}$$

Returning once more to equations 3.17 and including terms up to third order,

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + h^3 f'''(x) + \mathcal{O}\left(h^4\right) \\ f(x-h) &= f(x) - hf'(x) + \frac{1}{2}h^2 f''(x) - h^3 f'''(x) + \mathcal{O}\left(h^4\right), \end{aligned} \tag{3.21}$$

we see that in taking the sum, first and third order terms cancel, giving the *second central differences* expression for the second derivative with the same error order as first central differences,

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \mathcal{O}\left(h^2\right) \tag{3.22}$$

Likewise, a five-point formula for the second derivative can also be derived,

$$f''(x) = \frac{-f(x-2h) + 12f(x-h) - 30f(x) + 12f(x+h) - f(x+2h)}{12h^2} + \mathcal{O}\left(h^4\right) \tag{3.23}$$

again with error order h^4.

Boundaries

A limitation of this approach to reducing the error is that it relies on more and more neighboring points, which aren't available at the boundaries of our data set. This limitation is not a big deal for one-dimensional

problems, where the boundary is comprised of just two points, but in higher-dimensional problems the error at the boundary cannot be ignored. How can we we evaluate a central differences expression for a point with no left or right neighbor? If we are using three-point formulae, we could use the equivalent first forward or first backward expressions, but we know that they are less accurate (compare equations 3.16 and 3.18.

A solution for boundary points with the same error order as first central differences can be obtained by taking differences of $f(x+h), f(x+2h)$,

$$f(x+2h) - 4f(x+h) = -3f(x) - 2hf'(x) + \mathcal{O}\left(h^3\right), \tag{3.24}$$

and, rearranging,

$$f'(x) = -\frac{f(x+2h) - 4f(x+h) + 3f(x)}{2h} + \mathcal{O}\left(h^2\right), \tag{3.25}$$

which expression approximates the derivative at x using only x and points to the right of x.

Non-Uniform Grids

All formulae derived so far assumed that our data were distributed uniformly on a grid, but this is often not the case. It is easy enough to modify equations 3.21 to incorporate non-uniform spacings a, b,

$$\begin{aligned} b^2 f(x+a) &= b^2 f(x) + ab^2 f'(x) + \frac{1}{2}a^2 b^2 f''(x) + a^3 b^2 f'''(x) + \mathcal{O}\left(a^4\right) \\ a^2 f(x-b) &= a^2 f(x) - ba^2 f'(x) + \frac{1}{2}b^2 a^2 f''(x) - b^3 a^2 f'''(x) + \mathcal{O}\left(b^4\right), \end{aligned} \tag{3.26}$$

rearranging,

$$f'(x) = \frac{b^2 f(x+a) - a^2 f(x-b) + \left(b^2 - a^2\right) f(x)}{ab(b+a)} + \mathcal{O}\left(h^2\right) \tag{3.27}$$

where $h = \max(|a|, |b|)$, reducing to equation 3.18 when $a = b$. Notice that even-order terms no longer cancel exactly. This observation is particularly relevant when we consider the non-uniform expression for the second derivative (left as exercise to the reader). In that case, odd-order terms, most importantly the third-order terms, no longer cancel exactly, so the leading order error term is $\mathcal{O}(h)$ rather than $\mathcal{O}\left(h^2\right)$. In practice, the third-order derivatives almost cancel if $a \approx b$.

Richardson Extrapolation

If we can express a problem in the form,

$$A = A(h) + Kh^k + K'h^{k+1} + K''h^{k+2} + \ldots \tag{3.28}$$

where $h, k, A(h)$ are known, and the constants K^n are in general not known. Truncating at kth order in h, we get,

$$A = A(h) + Kh^k + \mathcal{O}\left(h^{k+1}\right). \tag{3.29}$$

By applying a technique known as *Richardson extrapolation*, we are able to rewrite this expression as

$$A = B(h) + \mathcal{O}\left(h^{k+1}\right), \tag{3.30}$$

i.e., we can generate a new expression for A with error one order higher than the original expression.

Halving the step size h,

$$A = A\left(\frac{h}{2}\right) + K\left(\frac{h}{2}\right)^k + \mathcal{O}\left(h^{k+1}\right) \tag{3.31}$$

and taking the difference $2^k \times (3.29) - (3.31)$,

$$\begin{aligned} \left(2^k - 1\right) A &= 2^k A\left(\frac{h}{2}\right) - A(h) + \mathcal{O}\left(h^{k+1}\right) \\ &= \frac{2^k A\left(\frac{h}{2}\right) - A(h)}{2^k - 1} + \mathcal{O}\left(h^{k+1}\right) \end{aligned} \tag{3.32}$$

Finally, defining

$$B(h) = \frac{2^k A\left(\frac{h}{2}\right) - A(h)}{2^k - 1}, \tag{3.33}$$

we arrive at equation 3.30.

Applying this method to the first central differences approximation,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(x) + \mathcal{O}\left(h^4\right), \tag{3.34}$$

by comparison with 3.29 we have

$$\begin{aligned} A &= f'(x) \\ A(h) &= \Delta_1(h) = \frac{f(x+h) - f(x-h)}{2h} \\ k &= 2 \\ K &= \frac{1}{6} f'''(x) \\ B(h) &= \frac{4\Delta_1\left(\frac{h}{2}\right) - \Delta_1(h)}{3} \end{aligned} \tag{3.35}$$

where we have used the common notation $\Delta_1(h)$ to denote the first central differences approximation. So our final approximation is,

$$f'(x) = \frac{4\Delta_1\left(\frac{h}{2}\right) - \Delta_1(h)}{3} + \mathcal{O}\left(h^4\right), \tag{3.36}$$

which, if you substitute in for Δ, you may find familiar. Applying Richardson extrapolation to equation 3.36, we get an expression with leading error order $\mathcal{O}\left(h^6\right)$ (the fifth order term cancels due to symmetry). This approach is useful not only for numerical differentiation, but also for deriving schemes to integrate of differential equations (*e.g.* Romberg integration).

Controlling Errors

A third limitation of the approaches so far is that they give no way to control the error in the calculation. We know that $\mathcal{O}\left(h^N\right)$ means that the absolute value of the error is at most some constant times $\left|h^N\right|$ when h is close enough to zero, but that constant could yet be too large for our purposes.

Consider the sequence:

$$f' \approx f'_k = \Delta_1\left(\frac{h}{2^k}\right). \tag{3.37}$$

As h tends to zero, f'_k tends to the true value of the derivative, and the difference between successive approximations $\left\|f'_k - f'_{k+1}\right\|$ tends to zero; we can iteratively reduce the magnitude of the error and get approximation arbitrarily close to the true value!

However, in practice, this is not possible: because of floating point approximations (not to mention other possible sources of error), we are limited in precision to

$$\left\| \frac{f'_k - f'_{k+1}}{f'_k + f'_{k+1}} \right\| \approx \epsilon \tag{3.38}$$

where ϵ is the machine epsilon (see section §2.4). However, we are also limited by precision loss in the operation $x \pm h$; in floating point arithmetic, `x-(x+h) != h`. When h is small compared to x, the error due to precision loss dominates the leading error in the approximation. For binary64 floats, the error due to precision loss in this operation has an approximate upper bound $\epsilon_p = 2^{-53} \cdot x/h$.

If we iterate for long enough, and h becomes sufficiently small, `x+h = x` and $\left\|f'_k - f'_{k+1}\right\| = 0$, an incorrect result. In the following code, lines 41–43 check if floating point underflow has occurred, raising a `ValueError` if appropriate.

```python
def first_forward_difference(f, x, h):
    return (f(x + h) - f(x)) / h


def first_backward_difference(f, x, h):
    return (f(x) - f(x - h)) / h


def first_central_difference(f, x, h):
    return (f(x + h) - f(x - h)) / 2 / h


def iterative_difference(f, x, h, tol, difference_function=first_central_difference):
    """

    Estimate the derivative of f at x using the specified difference function
    by iteratively reducing the initial step size h until
    | Delta(h) - Delta(h/2) | < tol .

    If the ratio h/x becomes so small that x+h==x, raise a ValueError reporting
    floating point underflow.

    :param f: function whose derivative is required
    :type f: callable, call signature f(x)
    :param x: point at which the function derivative is required
    :type x: float
    :param h: initial step size
    :type h: float
    :param tol: desired precision
    :type tol: float
    :param difference_function: function to calculate the derivative
    :type difference_function: callable, call signature g(f, x, h)
    :return: derivative to required precision, if found
    :rtype: float
    """
    f0 = difference_function(f, x, h)
    f1 = 1e10

    while abs(f0 - f1) > tol:
        h /= 2
        if x+h == x:
            raise ValueError('desired tolerance cannot be reached due to floating point underflow'
                             f' (current step size: {h})')
        f0, f1 = f1, difference_function(f, x, h)
    return f1
```

3.3 Integration

We can also use computers to accurately evaluate integrals which have no closed form. In this section we will consider solving equations of the form

$$I = \int_a^b f(x)\,dx. \tag{3.39}$$

All of the approaches we discuss will rely on the idea of the value of an integral being equal to a sum of discrete areas underneath the curve $f(x)$ between the points a and b.

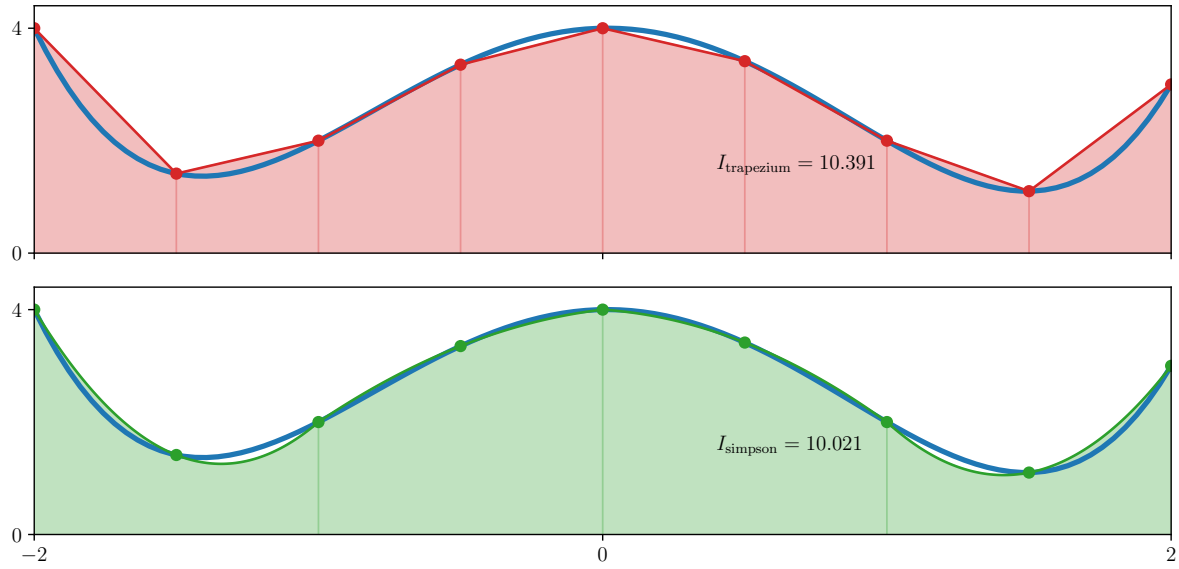

Figure 3.3: An illustration of the trapezium method (upper panel) and the Simpson method (lower panel) applied to $\frac{5x^4}{8} - \frac{x^3}{12} - \frac{21x^2}{8} + \frac{x}{12} + 4$ on the range $[-2, 2]$. The exact value of the integral is 10.

We will divide the area under the curve into n *slices* (c.f. figure 3.3), then approximate the areas of these slices, adding together their areas to give an approximation of the total area (in the limit of inifinitesimally thin slices, we by definition recover the exact value of the integral).

3.3.1 Newton-Cotes Formulae

The *Newton-Cotes formulae* are a class of integration schemes that promise to be exact for polynomials up to some order. The familiar *trapezium rule*, illustrated in figure 3.3, exact up to linear order, is one such scheme.

We begin by approximating the area of a single slice by a weighted sum of the values of f at m points $(x_0, \ldots, x_m)$; $x_i < x_{i+1}$,

$$A = \int_a^b f(x) \approx \sum_{i=1}^{m} w_i f(x_i) = \sum_{i=1}^{m} w_i f_i \tag{3.40}$$

where $f_i = f(x_i)$ and points x_1 and x_m are the endpoints of the slice.

For this expression to be exact for polynomials up to linear order, we must choose two points and be able to evaluate the following integrals exactly,

$$\begin{aligned} f(x) = 1 \to A &= \int_{x_1}^{x_2} dx = x_2 - x_1 \\ f(x) = x \to A &= \int_{x_1}^{x_2} x dx = \frac{1}{2}\left(x_2^2 - x_1^2\right) \end{aligned} \tag{3.41}$$

which requires us to solve the following pair of equations,

$$\begin{aligned} x_2 - x_1 &= w_1 + w_2; \\ \frac{1}{2}\left(x_2^2 - x_1^2\right) &= w_1 x_1 + w_2 x_2. \end{aligned} \tag{3.42}$$

The solution is,

$$w_1 = w_2 = \frac{x_2 - x_1}{2}, \tag{3.43}$$

which means the area of the slice is,

$$\begin{aligned} A &= \frac{x_2 - x_1}{2} f_1 + \frac{x_2 - x_1}{2} f_2 \\ &= (x_2 - x_1) \frac{(f_2 + f_1)}{2} \end{aligned} \tag{3.44}$$

i.e., the area of a trapezium with base $(x_2 - x_1)$ and heights f_1, f_2. Thus we approximate the value of the integral as the sum over the areas of the slices,

$$I = \int_a^b f(x)\, dx \approx \frac{1}{2} \sum_j^n (x_{j2} - x_{j1}) (f_{j2} + f_{j1}). \tag{3.45}$$

To go one order further, we need to consider three evenly-spaced points for each slice x_1, x_2, x_3 and the integrals,

$$\begin{aligned} f(x) = 1 \rightarrow A &= \int_{x_1}^{x_3} dx = x_3 - x_1 = w_1 + w_2 + w_3 \\ f(x) = x \rightarrow A &= \int_{x_1}^{x_3} x dx = \frac{1}{2}\left(x_3^2 - x_1^2\right) = w_1 x_1 + w_2 x_2 + w_3 x_3 \\ f(x) = x^2 \rightarrow A &= \int_{x_1}^{x_3} x^2 dx = \frac{1}{3}\left(x_3^3 - x_1^3\right) = w_1 x_1^2 + w_2 x_2^2 + w_3 x_3^2 \end{aligned} \tag{3.46}$$

The solution, left as an exercise to the reader, is *Simpson's rule*,

$$w_1 = w_3 = \frac{1}{3} h; \quad w_2 = \frac{4}{3} h \tag{3.47}$$

where $h = \frac{1}{2}(x_3 - x_1)$ Summarising, the value of the integral of a single slice is approximated by the sum,

$$A = \int_{x1}^{x3} f(x)\, dx \approx \frac{h}{3}(f_1 + 4f_2 + f_3) \tag{3.48}$$

The trapezium rule approximates successive segments of a curve as straight lines, Simpson's rule approximates successive segments as parabolae; higher order schemes approximate using higher order polynomials, and are thus exact for polynomials of the same order.

Weights can be worked out to arbitrary order and for arbitrarily-spaced points as the integral of *Lagrange basis polynomials*. Given m data points $(x_1, f_1), \ldots, (x_m, f_m)$, the Lagrange interpolation is given by

$$L(x) = \sum_{i=1}^{m} f(x_i)\, l_i(x) \tag{3.49}$$

where the $l_i(x)$ are Lagrange basis polynomials,

$$l_i(x) = \prod_{j \neq i}^{m} \frac{(x - x_j)}{(x_i - x_j)} \tag{3.50}$$

$L(x)$ is the unique $(m-1)$th order polynomial that passes through the m points (convince yourself that $L(x_i) = f_i$).

Given these definitions, the approximate area of a slice is,

$$A = \int_{x_1}^{x_m} f(x)\, dx \approx \int_{x_1}^{x_m} L(x)\, dx = \int_{x_1}^{x_m} \left(\sum_{i=1}^{m} f(x_i)\, l_i(x) \right) dx = \sum_{i=0}^{m} f(x_i) \underbrace{\int_{x_1}^{x_m} l_i(x)\, dx}_{w_i} \tag{3.51}$$

3.3.2 Romberg's Method

Applying Richardson extrapolation to the trapezium rule yields Simpson's rule. Further application leads to *Boole's rule*, a Newton-Cotes formula that is exact to fifth order. This approach is known as *Romberg's method*.

Beyond Boole's rule, the Newton-Cotes formulae and the Romberg interpolants differ. At higher orders, the former become unstable, containing large weights of different signs, and the error grows exponentially for large order; Romberg's method is relatively stable.

3.3.3 Error

Using Taylor series, we can construct an exact expression for an arbitrary integral between points x_0, x_1, with spacing $2h$ and midpoint x,

$$\begin{aligned} A^{\text{exact}} &= \int_{-h}^{h} f(x+y)\,dy \\ &= \int_{-h}^{h} \sum_{n=0}^{\infty} f^n(x)\,\frac{y^n}{n!} \\ &= \sum_{n=0}^{\infty} f^n(x) \int_{-h}^{h} \frac{y^n}{n!} \\ &= \sum_{n=0}^{\infty} f^n(x) \left[\frac{y^{n+1}}{(n+1!)}\right]_{-h}^{h} \\ &= 2\sum_{n\ \text{even}}^{\infty} f^n(x)\,\frac{h^{n+1}}{(n+1!)} \\ &= 2hf(x) + \frac{2}{3!}h^3 f''(x) + \mathcal{O}\left(h^5\right) \end{aligned} \tag{3.52}$$

where all the odd-order terms vanish due to symmetry.

Consider now the area, A^{est}, of a single trapezium (base $2h$) in an application of the trapezium rule,

$$\frac{A^{\text{est}}}{h} = f(x+h) + f(x-h) \tag{3.53}$$

The Taylor expansion of the RHS is well known to us,

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + h^3 f'''(x) + \mathcal{O}\left(h^4\right) \\ f(x-h) &= f(x) - hf'(x) + \frac{1}{2}h^2 f''(x) - h^3 f'''(x) + \mathcal{O}\left(h^4\right), \end{aligned} \tag{3.21}$$

Rearranging, and substituting into 3.53,

$$A^{\text{est}} = 2hf(x) + h^3 f''(x) + \mathcal{O}\left(h^5\right) \tag{3.54}$$

Comparing with the exact expression,

$$\begin{aligned} A^{\text{est}} - A^{\text{exact}} = \Delta A &= \left(1 - \frac{2}{3!}h^3\right) f''(x) + \mathcal{O}\left(h^5\right) \\ &= \frac{2}{3}h^3 f''(x) + \mathcal{O}\left(h^5\right) \end{aligned} \tag{3.55}$$

This is the error for a single trapezium; the value of the integral is a sum over many trapezia,

$$I^{\text{est}} = \sum_{i=0}^{n} A_i^{\text{est}}, \tag{3.56}$$

and so the total error ΔI is approximately

$$\Delta I \approx \frac{2}{3}nh^3 f''(x) = \frac{1}{3}(b-a)\,h^2 \langle f'' \rangle \tag{3.57}$$

where a, b is interval of the integration and $\langle f'' \rangle$ represents an average of the second derivative on the interval.

The preceding analysis shows that the trapezium rule approximation has an error order h^2; Simpson's method and Boole's rule improve the order to h^4 and h^6 respectively.

3.3.4 Simpson's Method from Taylor Series

Returning to the exact Taylor series expansion of the integral,

$$A^{\text{exact}} = 2hf(x) + \frac{2}{3!}h^3 f''(x) + \mathcal{O}\left(h^5\right), \tag{3.52}$$

we see that the second term is the second derivative of the integrand. What form does it take? In general, we don't know, but we have learned a lot of methods to approximate function derivatives—let's apply one.

Recall the second central differences approximation,

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \mathcal{O}\left(h^2\right); \tag{3.22}$$

substituting into equation 3.52 with step size k, we get

$$A^{\text{exact}} = 2hf(x) + \frac{2}{3!}h^3\left(\frac{f(x+k) - 2f(x) + f(x-k)}{k^2} + \mathcal{O}\left(k^2\right)\right) + \mathcal{O}\left(h^5\right). \tag{3.58}$$

Setting $k = h$, this expression simplifies to,

$$\begin{aligned} A^{\text{exact}} &= 2hf(x) + \frac{1}{3}h\left(f(x+h) - 2f(x) + f(x-h)\right) + \mathcal{O}\left(h^5\right) \\ &= \frac{f(x+h) + 4f(x) + f(x-h)}{3}, \end{aligned} \tag{3.59}$$

which is Simpsons' method.

3.3.5 Gaussian Quadrature

In §3.3.1 we discussed deriving integration schemes involving n points that are exact for order $n-1$ polynomials. Our starting point was the integral over a slice,

$$A = \int_{x_1}^{x_m} f(x)\,dx, \tag{3.39}$$

with arbitrary end points (x_1, x_m). However, by fixing the end points to $(-1, 1)$, we automatically generate approximations that are exact for all odd-order monomials! In this way, with n points we can construct an approximation that is exact for any order-$2n-1$ polynomial.

Analogously with the Newton-Cotes formulae, we begin by approximating the area of a single slice by a weighted sum of the values of f at m points $(x_0, \ldots, x_m)$; $x_i < x_{i+1}$,

$$A = \int_{-1}^{1} f(x) \approx \sum_{i=1}^{m} w_i f(x_i) = \sum_{i=1}^{m} w_i f_i. \tag{3.60}$$

For two points, in this context called *nodes*, x_1, x_2; $x_1 < x_2$ somewhere on the interval $(-1, 1)$, we have four equations with four unknowns,

$$\begin{aligned} f(x) = 1 \to A &= \int_{-1}^{1} dx = 2 = w_1 + w_2 \\ f(x) = x \to A &= \int_{-1}^{1} x dx = 0 = w_1 x_1 + w_2 x_2 \\ f(x) = x^2 \to A &= \int_{-1}^{1} x^2 dx = \frac{2}{3} = w_1 x_1^2 + w_2 x_2^2 \\ f(x) = x^3 \to A &= \int_{-1}^{1} x^3 dx = 0 = w_1 x_1^3 + w_2 x_2^3 \end{aligned} \tag{3.61}$$

Solving, we find,

$$x_2 = -x_1 = \frac{1}{\sqrt{3}}; \quad w_1 = w_2 = 1. \tag{3.62}$$

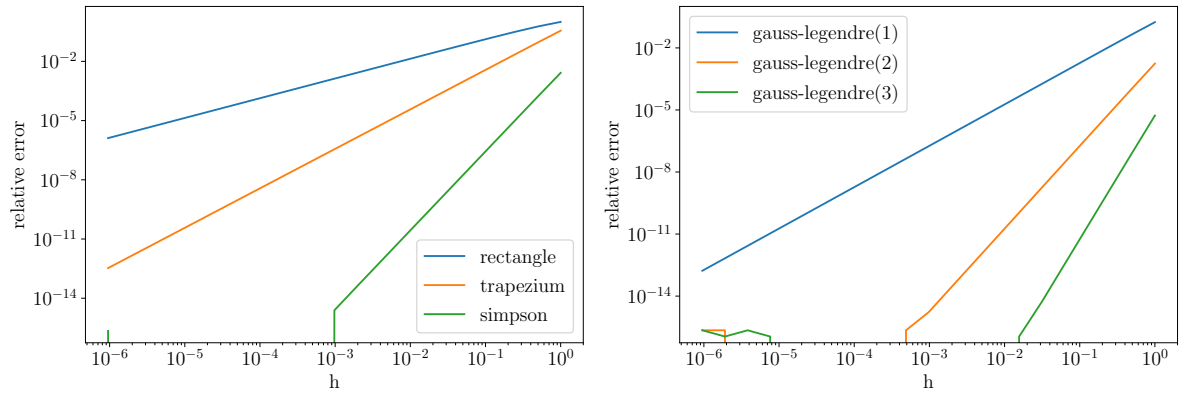

Figure 3.4: Left panel: relative error in the integral $\int_0^1 xe^x dx$ as a function of step size, h, for the three lowest-order Newton cotes methods. Right panel: the same for the three lowest-order Gauss-Legendre quadratures. At the same order, Gauss-Legendre quadratures converge two orders of h faster than Newton-Cotes methods.

To this level of approximation, the area of a slice is thus,

$$A \approx f\left(-1/\sqrt{3}\right) + f\left(+1/\sqrt{3}\right). \tag{3.63}$$

We can improve upon this result by solving for more nodes,

$$\sum_{i=1}^{m} w_i x_i^n = \begin{cases} \frac{2}{n-1} & n \text{ even;} \\ 0 & n \text{ odd,} \end{cases} \tag{3.64}$$

for $0 \leq n < N$.

Typically, the limits of the integral of interest are not $[-1, 1]$; in the general case, $[a, b]$, we must first apply the coordinate transformation,

$$\begin{aligned} t &= \frac{b-a}{2}x + \frac{b+a}{2} \\ \int_a^b f(t)\, dt &= \int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right)\frac{b-a}{2}dx \\ &= \int_{-1}^{1} g(x)\, dx \end{aligned} \tag{3.65}$$

The approximate expression for the integral the becomes,

$$\int_a^b f(t)\, dt \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{b+a}{2}\right) \tag{3.66}$$

Strictly speaking, the method we have presented in this section is *Gauss-Legendre quadrature*. Just as the Newton-Cotes rules approximate the integrand using Lagrange basis polynomials, the Gauss-Legendre quadrature approximates the integrand with *Legendre polynomials*. Legendre polynomials are used for integrals over the closed interval $[-1, 1]$. In situations were there are singularities at the end-points of the integration, or when the range is not finite, other polynomials are used.

Figure 3.4 compares the relative errors of Newton-Cotes methods and Gauss-Legendre quadrature. For the same number of points per slice, the Gauss-Legendre quadrature has an error term h^2 times that of the Newton-Cotes method. For n points, Gauss-Legendre quadrature has a leading error term $\mathcal{O}\left(h^{2n}\right)$.

Chapter 4

Local Optimisation

The problem of function optimisation has applications in fields as varied as circuit design and economics, as well as in the natural sciences. For example, in business, a problem of optimally allocating resources requires the minimisation of some cost function with respect to some parameters; in chemical physics, predicting the ground state of an assembly of atoms interacting *via* some interaction potential is a problem of finding the lowest point on the potential energy surface.

In this chapter we discuss the problem of finding stationary points of functions, *i.e.*, minima, maxima, or saddle points. Since the problem of locating a stationary point of a function is equivalent to finding zeros of that function's first derivative, in §4.1 we begin our discussion with methods for finding zeros of functions. We then adapt some of these methods for finding local minima/maxima of functions of one variable (§4.2) and then functions of many variables (§4.3). We round off our discussion of local optimisation in §4.4 by exploring methods for finding first-order saddle points of functions.

4.1 Roots of Equations

The problem of finding zeroes, or roots, of an equation is more general that it would appear at first glance. Indeed, any equation $f(x) = g(x)$ can be recast as a root-finding problem by rearranging to $f(x) - g(x) = 0$.

Sometimes such equations can be solved analytically; for example, there are general solutions for the roots of polynomials of degree ≤ 4. However, almost all polynomials degree ≥ 5 admit of no analytical solution, the simplest being,

$$x^5 - x - 1 = 0, \tag{4.1}$$

which has one real, irrational root, $x \approx 1.1673$.

How do we go about finding this root? There are many methods available to tackle this problem, and in the following sections we give a brief survey. The methods will consider all rely on making an initial guess for the solution, x_0, and iteratively improving upon that guess until some convergence criterion is satisfied.

4.1.1 Convergence

We have a function, $f(x)$, an equation, $f(x) = 0$, and a solution, $x = \xi$; We require a second function, $\phi(x)$, a second equation, $x = \phi(x)$, with the same solution, $x = \xi$, and sequence

$$x_{k+1} = \phi(x_k), \tag{4.2}$$

such that,

$$\lim_{k\to\infty} \phi(x_k) = \xi. \tag{4.3}$$

In order for this sequence to converge, we require the absolute error $|\epsilon_k| = |\xi - x_k|$ to decrease after each iteration. Starting with,

$$\xi + \epsilon_{k+1} = x_{k+1} = \phi(x_k) = \phi(\xi + \epsilon_k), \tag{4.4}$$

we expand the right-most term as a Taylor series,

$$\begin{aligned} \xi + \epsilon_{k+1} &= \phi(\xi) + \epsilon_k \phi'(\xi) + \mathcal{O}\left(\epsilon_k^2\right) \\ \epsilon_{k+1} &= \epsilon_k \phi'(\xi) + \mathcal{O}\left(\epsilon_k^2\right) \end{aligned} \tag{4.5}$$

therefore a necessary (but not sufficient) condition for the sequence to converge to ξ is $\|\phi'(\xi)\| < 1$.

However, a promise of convergence is no good to us unless it can be delivered in a finite, hopefully short, time. How quickly does $\phi(x)$ converge to ξ? In general we can write,

$$\lim_{k->\infty} \frac{|\epsilon_{k+1}|}{|\epsilon_k|^q} = \mu \tag{4.6}$$

where q is the *order of convergence* and μ is the *rate of convergence* (in fact, this is a relatively strict definition of convergence called *Q-convergence*; other definitions exist). Comparing equations 4.5 and 4.6, the order of convergence q is one, there is *first-order convergence*, and the rate of convergence μ is equal to $\phi'(\xi)$.

If the first derivative of ϕ where to vanish at ξ, but not the second, *i.e.*, $\phi'(\xi) = 0$, $\phi''(\xi) \neq 0$, then,

$$\epsilon_{k+1} \approx \frac{\epsilon_k^2}{2} \phi''(\xi) \tag{4.7}$$

and the method would exhibit second-order convergence with rate $\phi''(\xi)/2$.

A practical method to approximate the order of convergence is

$$q \approx \frac{\log \left| \frac{x_{k+1} - x_k}{x_k - x_{k-1}} \right|}{\log \left| \frac{x_k - x_{k-1}}{x_{k-1} - x_{k-2}} \right|} \tag{4.8}$$

4.1.2 Rearrangement

One approach is to solve for roots of $f(x) = 0$ is to rearrange the equation into a slowly-varying function of x,

$$x = \phi(x), \tag{4.9}$$

and then, choosing some initial value x_0 close to the root, generate the recurrence relation,

$$x_{k+1} = \phi(x_k), \tag{4.10}$$

and hope that it converges to a root in a small number of iterations.

In the case of our unsolvable quintic polynomial, this process amounts to evaluating the recurrence relation,

$$x_{k+1} = \sqrt[5]{x_k + 1}, \tag{4.11}$$

until $|x_{k+1} - x_k| = |\epsilon_k| < \epsilon_{max}$.

Does our rearranged quintic converge to a root?

$$\begin{aligned} \phi(x) &= \sqrt[5]{x+1} \\ \phi'(x) &= -\frac{\sqrt[5]{x+1}}{5(x+1)} \\ \epsilon_{k+1} &= -\epsilon_k \frac{\sqrt[5]{x+1}}{5(x+1)} + \mathcal{O}\left(\epsilon_k^2\right) \\ \lim_{k->\infty} \frac{|\epsilon_{k+1}|}{|\epsilon_k|} &= \frac{\sqrt[5]{x+1}}{5(x+1)} \end{aligned} \tag{4.12}$$

Since $\phi'(\xi) \approx 0.1 < 1$, the method converges, with order 1 and rate ≈ 0.1. With an initial guess $x_0 = 0$, convergence to within $\epsilon_{max} = 10^{-8}$ of the true value is reached in just 9 iterations.

Question 8 Rearrangements

We could have rearranged 4.1 in a second way, namely to the recurrence relation,

$$x = \phi(x) = x^5 - 1. \tag{4.13}$$

Does the method converge with this choice of ϕ?

4.1.3 Bisection

One the conceptually simplest methods for finding zeros of a one-dimensional function is the *bisection method*. Beginning with a function $f(x)$ and an interval that brackets a zero, *i.e.*, (a, b) such that $f(a) f(b) <= 0$, we successively choose the more narrow interval (a, c) or (c, b) where $c = (a + b)/2$. If $f(a) f(c) <= 0$, then $f(c) f(b) > 0$, so we choose (a, c) as our next interval. Otherwise, $f(c) f(b) <= 0$, so we choose (c, b) We repeat the process until the width of the interval $b - a$ is less than some tolerance.

We can write the recurrence relation as,

$$a_{k+1}, b_{k+1} = \phi(a_k, b_k) = \begin{cases} \left(a_k, \frac{a_k+b_k}{2}\right) & f\left(\frac{a_k+b_k}{2}\right) > 0; \\ \left(\frac{a_k+b_k}{2}, b_k\right) & f\left(\frac{a_k+b_k}{2}\right) < 0. \end{cases} \tag{4.14}$$

Convergence is guaranteed with linear order and rate $1/2$, but faster methods are available.

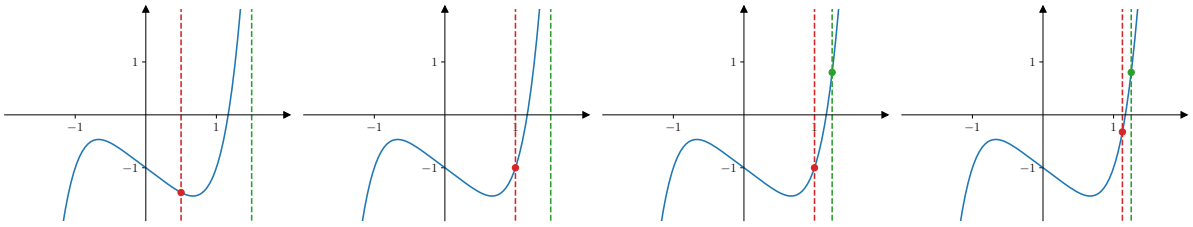

Figure 4.1: Four steps of the bisection method applied to $x^5 - x - 1$ with starting interval $[0.5, 1.5]$.

4.1.4 Secant Method

The secant method approaches the root of a curve by making a linear approximation to the curve, and finding the root of the line.

Given a curve and two points (x_0, x_1) that bracket a zero, we find the secant that intersects the curve at those points, which is given by the equation,

$$y = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1) \tag{4.15}$$

Rearranging, we find that the root of the line, x_2, is at the point,

$$x_2 = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)} \tag{4.16}$$

We repeat the process, replacing $(x_0, x_1) \leftarrow (x_1, x_2)$, until $|x_1 - x_0| < \epsilon_{max}$.

The secant method has an order of convergence equal to the golden ratio, $\phi = \frac{1+\sqrt{5}}{2}$, and as such is faster than bisection. However, it fails to find multiple roots, and may fail to find a root if there is a stationary point on the bracketing interval.

Convergence of the Secant Method The error in the $k+1$th iterate is,

$$\begin{aligned} x_{k+1} &= x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}, \\ x_{k+1} &= \frac{x_{k-1}f(x_k) - x_k f(x_{k+1})}{f(x_k) - f(x_{k+1})}, \\ \epsilon_{k+1} &= \frac{\epsilon_{k-1}f(x_k) - \epsilon_k f(x_{k+1})}{f(x_k) - f(x_{k+1})}. \end{aligned} \tag{4.17}$$

By the mean value theorem, there exists some $\eta_k \in (x_k, \xi)$ such that,

$$\begin{aligned} f'(\eta_k) &= \frac{f(x_k) - f(\xi)}{x_k - \xi}, \\ f(x_k) &= \epsilon_k f'(\eta_k), \\ f(x_{k-1}) &= \epsilon_{k-1} f'(\eta_{k-1}). \end{aligned} \tag{4.18}$$

Substituting back in to the expression for the error in the $k+1$th iterate,

$$\begin{aligned} \epsilon_{k+1} &= \epsilon_k \epsilon_{k-1}\frac{f'(\eta_k) - f'(\eta_{k-1})}{f(x_k) - f(x_{k-1})}, \\ \epsilon_{k+1} &\propto \epsilon_k \epsilon_{k-1}. \end{aligned} \tag{4.19}$$

However, from the definition of q-convergence,

$$\begin{aligned} &\epsilon_{k+1} \propto \epsilon_k^q, \\ &\implies \epsilon_{k+1} \propto \epsilon_k^{1+1/q}, \\ &\implies q = 1 + 1/q, \\ &\implies q = \frac{1+\sqrt{5}}{2}, \end{aligned} \tag{4.20}$$

since $q > 0$ by definition.

4.1.5 The Newton–Raphson method

The *Newton–Raphson method*, like the secant method, is based on a linear approximation of a curve close to a zero, but in contrast to the secant method, finds the root of the tangent to a curve at a point, rather than that of a secant between two points.

Naturally, this approach requires us to evaluate not only the function, but also its derivatives.

We can arrive at the Newton–Raphson recurrence relation by writing the equation of the tangent to the curve $f(x)$ as x_k,

$$y = f'(x_k)(x - x_k) + f(x_k). \tag{4.21}$$

Solving for the root of the tangent, x_{k+1}, our next iterate is,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{4.22}$$

This process is repeated until the difference between successive guesses is less than some tolerance.

The secant method is sometimes called the *discrete Newton method*, since it is the same as approximating the first derivative by the two-point formula. In spite of this similarity, the Newton–Raphson method boasts superior convergence properties to the secant method, converging quadratically ($p = 2$) in favourable conditions. However, many unfavourable conditions are possible. If, at a root, the first derivative is discontinuous or does not exist, it cannot be found. If the first derivative is zero, the root can be found, but not with quadratic convergence. For some functions at some starting points, such as our cryptic quintic at $x = 0.5$, the iterates form a cycle and never converge (try it).

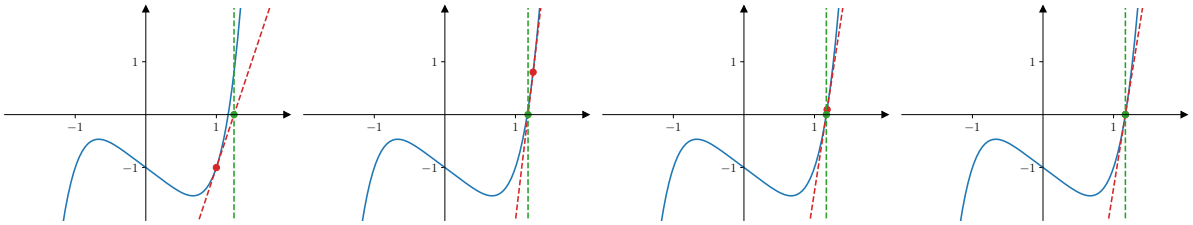

Figure 4.2: Four steps of Newton-Raphson applied to $x^5 - x - 1$ starting at $x = 1$.

Convergence of Newton–Raphson The Newton–Raphson update and its first two derivatives are,

$$\begin{aligned} \phi_{\text{NR}}(x) &= x - \frac{f(x)}{f'(x)}, \\ \phi'_{\text{NR}}(x) &= \frac{f''(x) f(x)}{(f'(x))^2}, \\ \phi''_{\text{NR}}(x) &= \frac{f''(x)}{f'(x)}. \end{aligned} \tag{4.23}$$

Recalling the expression for the error in the $k+1$th iterate in equation 4.4,

$$\begin{aligned} \epsilon_{k+1} &= \epsilon_k \phi'_{\text{NR}}(\xi) + \frac{\epsilon_k^2}{2} \phi''_{\text{NR}}(\xi) + \mathcal{O}\left(\epsilon_k^3\right), \\ &= \frac{\epsilon_k^2}{2} \frac{f''(\xi)}{f'(\xi)} + \mathcal{O}\left(\epsilon_k^3\right), \\ \lim_{k->\infty} \frac{|\epsilon_{k+1}|}{|\epsilon_k|^2} &= \frac{1}{2} \frac{f''(\xi)}{f'(\xi)}; \end{aligned} \tag{4.24}$$

therefore, close to the root, the Newton-Raphson method converges (or diverges) with order $q = 2$ and rate $\mu = \frac{1}{2} \frac{f''(\xi)}{f'(\xi)}$.

4.1.6 Inverse quadratic interpolation

We can also approximate the curve with higher-order polynomials. In *inverse quadratic interpolation*. Given a function $f(x) = y$, the idea is to approximate the inverse of f, $f^{-1}(y) = x$. As the name suggests, inverse quadratic interpolation approximates f^{-1} with a parabola, specifically a *Lagrange polynomial*.

Given three points (x_0, x_1, x_2), the parabola is given by

$$f^{-1}(y) = \sum_{i=0}^{2} x_i \prod_{i \neq j} \frac{y - f(x_j)}{f(x_i) - f(x_j)} \tag{4.25}$$

The next approximation to the zero of f is then identified by computing $x_3 = f^{-1}(0)$,

$$x_3 = f^{-1}(0) = \sum_{i=0}^{2} x_i \prod_{i \neq j} \frac{f(x_j)}{f(x_i) - f(x_j)} \tag{4.26}$$

With this new point, our three points are updated $(x_0, x_1, x_2) = (x_1, x_2, x_3)$, and the process is repeated until the range of (x_0, x_1, x_2) is less than some tolerance.

The recurrence relation for the method is,

$$x_{n+1} = \frac{f_{n-1} f_n}{(f_{n-2} - f_{n-1})(f_{n-2} - f_n)} x_{n-2} + \frac{f_{n-2} f_n}{(f_{n-1} - f_{n-2})(f_{n-1} - f_n)} x_{n-1} + \frac{f_{n-2} f_{n-1}}{(f_n - f_{n-2})(f_n - f_{n-1})} x_n \tag{4.27}$$

where $f_i = f(x_i)$.

In favourable cases, this method converges faster than the bisection method, but pathological cases, *e.g.* any two of $(f(a), f(b), f(c))$ being equal, convergence cannot be achieved.

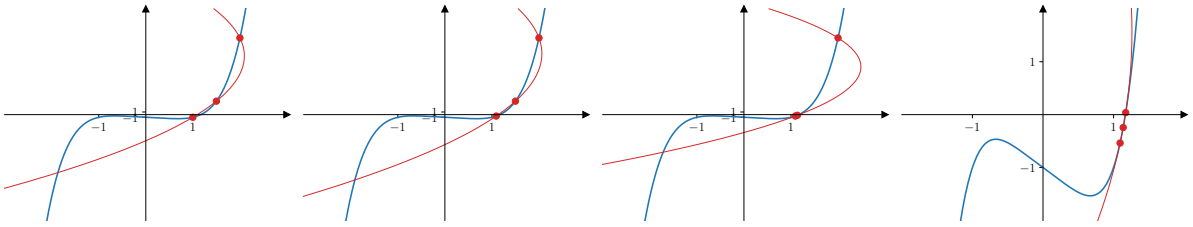

Figure 4.3: Four steps of inverse parabolic interpolation applied to $x^5 - x - 1$ with starting interval $[1, 2]$.

Question 9 Pathological Cases

Implement the five root-finding methods we have studied, and use them to find the roots of,

1. $x^5 - x - 1 = 0$;
2. $16x^4 - 8x + 3 = 0$;
3. $x^3 - 2x^2 - 11x + 12 = (x - 4)(x - 1)(x + 3) = 0$,

for a range of starting values (plot the curves to get an idea of what a sensible range of values might be). What do you notice? Do all of the methods find all of the roots?

Estimate the order of convergence using the formula given above. Is the order of convergence always the theoretical maximum?

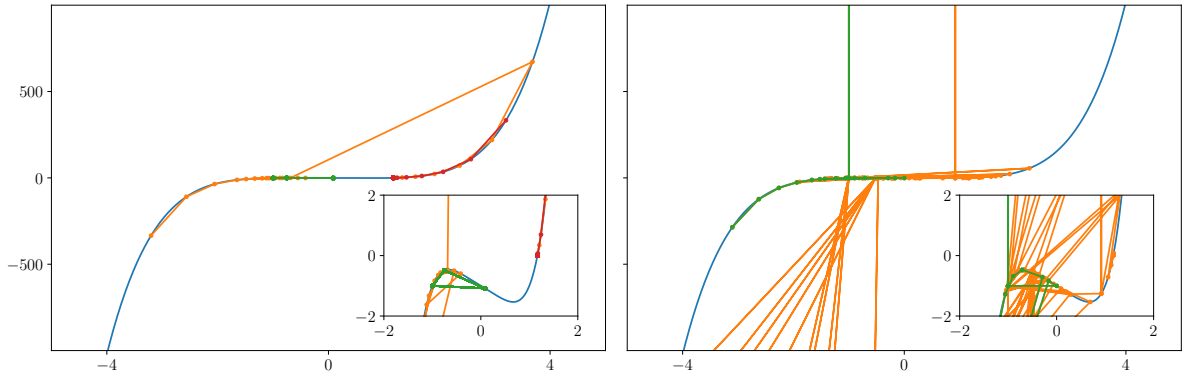

Figure 4.4: Pathological inputs for the Newton-Raphson (left) and secant (right) methods applied to the quintic polynomial. Left: orange, green, and red curves correspond to trajectories starting at x_0 -4,0,and 4, respectively. $x_0 = 4$ is close enough to the root for the method to converge smoothly to the root. However, there is a stationary point between $x_0 = 0$ and the only real root, so the method falls into a three-element cycle. A trajectory starting at $x_0 = -4$ is able to jump to the other side of the root, overcoming this problem. Right: the secant method starting at $(-4.5, -3.5)$ converges prematurely to $x \approx -1$ if the tolerance is set too high (10^{-8}) due to catastrophic cancellation. Reducing the tolerance to 10^{-12} avoids this problem. Notice that, from this starting point, it would be impossible to converge to the root using binary32 floats.

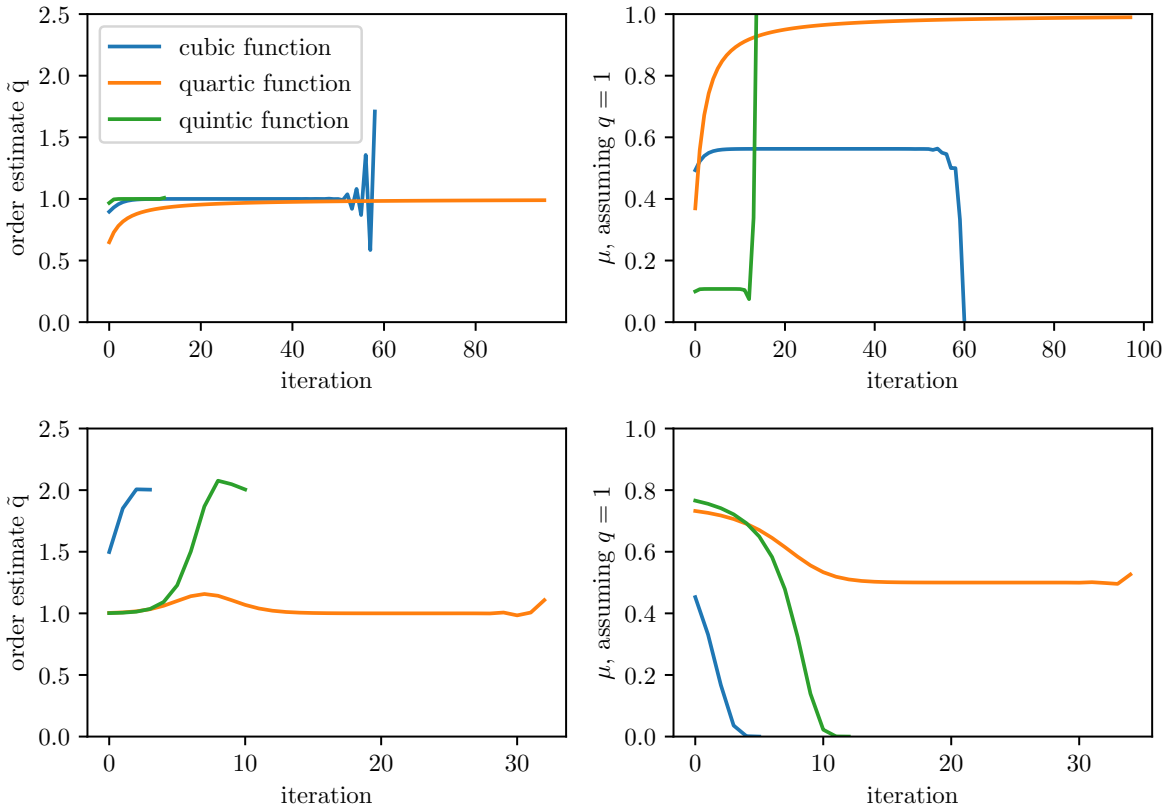

Figure 4.5: Estimated rates (left) and orders (right) of convergence of the rearrangement (upper) and Newton-Raphson (lower) methods applied to the cubic, quartic, and quintic polynomials (blue, orange, and green curves, respectively).

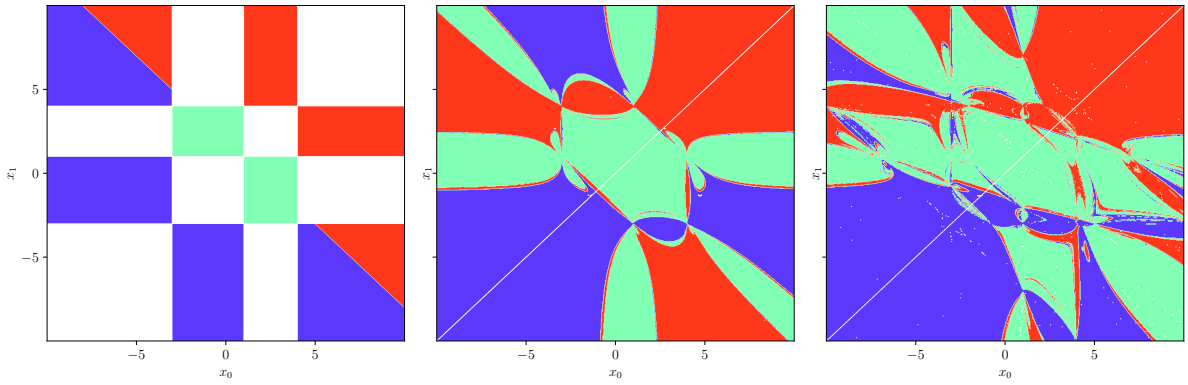

Figure 4.6: Basins of attraction for the bisection (left), secant (middle), and inverse quadratic interpolation (right) methods applied to the cubic polynomial. The roots -3, 1, and 4 are represented by the colours blue, green, and red, respectively. If the input is invalid for the method, or the method did not converge to a root, the point is plotted in white. Notice that, even when the secant and inverse quadratic interpolation methods begin at a point that brackets a root, they do not necessarily converge to that root.

4.2 Stationary points of functions of a single variable

The problem of finding a stationary point of a function $f(x)$ is equivalent to finding roots of its first derivative, *i.e.*, ξ such that $f'(\xi)$. As such, in this section we will encounter some optimisation methods that are quite similar to those we saw in the previous section.

4.2.1 The golden-section method

The derivative-free bisection method described in section §4.1.3 required us to first define a *bracket*, an interval (a, b), within which a root was certain to be found, *i.e.*, such that $f(a) f(b) <= 0$. Bracketing a minimum (or maximum) requires an extra point $c \in (a, b)$ (a *probe point*) on an interval (a, b) with the constraint that $f(a) > f(c) < f(b)$. If such a value c exists, then there exists at least one minimum on the interval (possibly more). With these conditions met, it is easy to imagine extending the section method for roots to a bisection method for extrema; choose a second point, $d \in (a, b)$ and construct a new, narrower bracket from (a, b, c, d) discarding one of a, b depending of the relative values of $f(c)$ and $f(d)$.

We could choose any two points $c, d \in (a, b)$ and, if the bracketing condition is met, the algorithm will eventually converge. Say we have a bracketing interval (a, b) containing two probe points $c < d$. If $f(c) < f(d)$, our new bracketing interval is (a, d); otherwise, the new interval is (c, b). In general, the widths of those intervals are not the same, *i.e.*, $|b - c| \neq |d - a|$. If we are lucky, we always choose the narrower interval, and we converge quickly. If we have a run of bad luck, we choose the wider interval every time, and converge slowly (*e.g.*, figure 4.7, left). To ensure reliable convergence, we should choose c, d so that $|b - c| = |d - a|$. One way to do this is to choose probe points c, d such that

$$c = b - \frac{b - a}{x}; \quad d = a + \frac{b - a}{x} \tag{4.28}$$

for $x > 0$ and $x \neq 2$. This approach has the advantage of producing a fixed relative reduction in the interval width, but in general requires us to evaluate the function twice every step, compared with just once in the previous scheme (see figure 4.7 for the example case $x = 1/3$.

To ensure a fixed reduction in the interval width, and to ensure that the function is evaluated only once per step, we require that both points c, d define the new bracket, *i.e.*,

$$\frac{b - a}{d - a} = \frac{d - a}{c - a}. \tag{4.29}$$

Solving, we find that x is equal to our old friend, the golden ratio. Choosing probe points using equation 4.28 setting $x = \left(1 + \sqrt{5}\right)/2$ is known as the *golden-section method*. The right-most panel of figure 4.7 demonstrates the narrowing of the interval width for a golden section search.

Whatever the scheme, we iterate until the width of the bracket $|b - a|$ (the error estimate) is less than some tolerance.

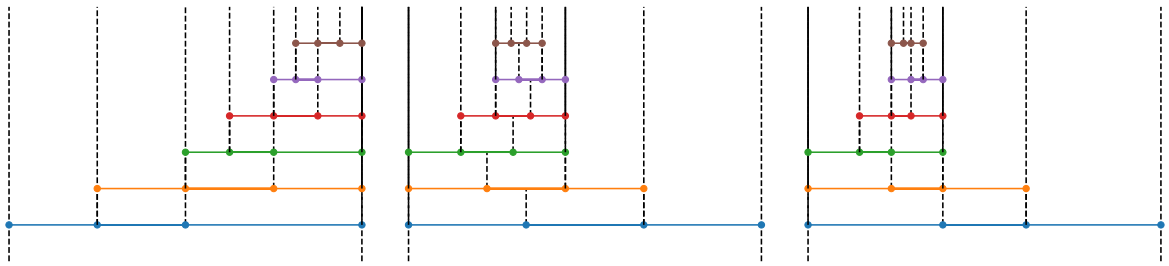

Figure 4.7: Interval width of the course of an optimisation for three different update schemes. Left: Starting with three points $a < c < b$, choose a fourth point d at the centre of the widest sub-interval $(a, c), (c, b)$. With these four points, there are two possible updates to the bracket, with different widths. In the case of a run of bad luck (pictured), we always choose the wider option. Centre: Starting with two points a, b, choose c, d $1/3$ and $2/3$ along (a, b). The two possibilities for the next bracket, a, c, d and c, d, b are of equal width, but we have to evaluate the function twice at each step. Right: A golden section search. At each step, the function is evaluated once, and the width of the bracket is reduced by a fixed amount.

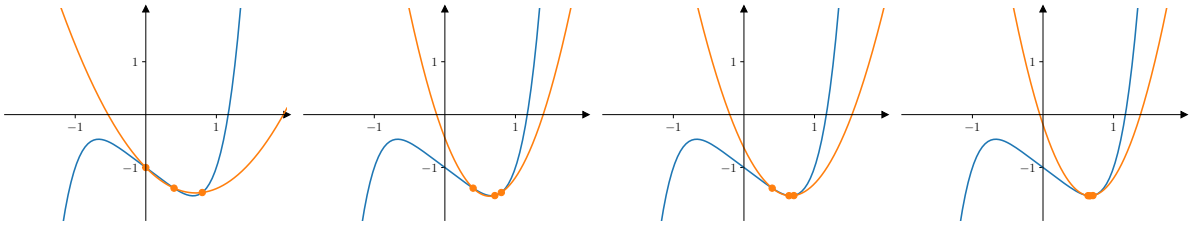

Figure 4.8: Successive parabolic interpolation applied to the quintic with $(a, b, c) = (0.0, 0.4, 0.8)$. What happens if you set $(a, b, c) = (0.0, 0.5, 1.0)$?

4.2.2 Successive parabolic interpolation

As with inverse quadratic interpolation, we can locate extrema by fitting parabolas. Given three points a, b, c, we find the unique parabola that passes through them, and then find the extremum of the parabola, d. We then iterate, setting $(a, b, c) \leftarrow (b, c, d)$, until the range of (a, b, c) is below some tolerance. This method is known as *successive parabolic interpolation*, and has an order of convergence $q \approx 1.324$.

At each step, we find the Lagrange polynomial for a, b, c,

$$p(x) = \frac{(x-b)(x-c)}{(a-b)(a-c)} f(a) + \frac{(x-a)(x-c)}{(b-a)(b-c)} f(b) + \frac{(x-a)(x-b)}{(c-a)(c-b)} f(c) \tag{4.30}$$

and its extremum, d,

$$d = \frac{1}{2} \frac{a g_a + b g_b + c g_c}{g_a + g_b + g_c}, \tag{4.31}$$

where,

$$g_a = a(f(c) - f(b)); \quad g_b = b(f(a) - f(c)); \quad g_c = c(f(b) - f(a)). \tag{4.32}$$

A few steps of successive parabolic interpolation applied to our quintic function are shown in figure 4.8.

The three points a, b, c need not bracket a minimum or a maximum, and the method can converge to either type of stationary point. If we want to force the method toward a minimum, we can start the iteration with a known bracket a, c, b, find the new point d, and choose a new bracket according to the following rules,

$$(a, c, b) = \begin{cases} (a, d, c) & d < c, \quad f(d) < f(c) \\ (d, c, b) & d < c, \quad f(d) > f(c) \\ (c, d, b) & d > c, \quad f(d) < f(c) \\ (a, c, d) & d > c, \quad f(d) > f(c). \end{cases} \tag{4.33}$$

4.2.3 Newton–Raphson

The Newton–Raphson method for finding minima is closely related to root-finding version, and like successive parabolic interpolation also involves fitting parabolas.

Given a twice-differentiable function $f(x)$ and a starting point x_0, we find the parabola $p(x) = ax^2 + bx + c$ such that,

$$p(x_0) = f(x_0); \quad p'(x_0) = f'(x_0); \quad p''(x_0) = f''(x_0); \tag{4.34}$$

our new point x_1 is then the extremum of this parabola.

We can work out the expression for the update geometrically, or derive it from a Taylor series expansion. Let the coordinate of the minimum be ξ, and the approximation at iteration k be $x_k = \xi + \epsilon_k$; then,

$$f(\xi) = f(x_k + \epsilon_k) = f(x_k) + \epsilon_k f'(x_k) + \mathcal{O}(\epsilon_k^2). \tag{4.35}$$

Taking the first derivative w.r.t x_k,

$$f'(xi) = f'(x_k) + \epsilon_k f''(x_k) + \mathcal{O}(\epsilon_k^2). \tag{4.36}$$

At a stationary point, the first derivative is zero, so

$$\epsilon_k = -\frac{f'(x_k)}{f''(x_k)} + \mathcal{O}(\epsilon_k^2) \tag{4.37}$$

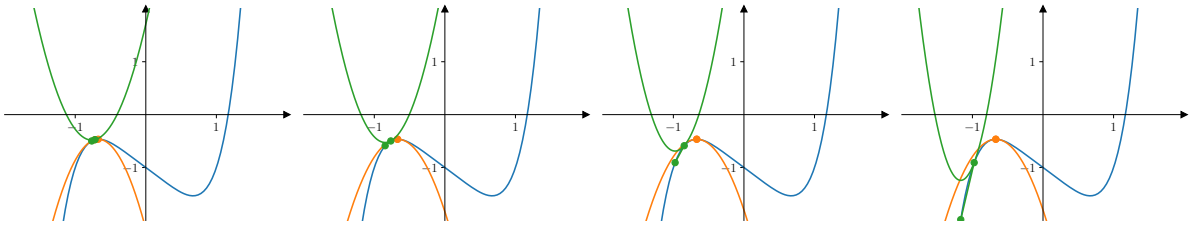

Figure 4.9: The first four iterations of Newton–Raphson optimisation applied to the quintic polynomial starting at $x_0 = -0.72$. The orange curve follows the unmodified trajectory, whereas the green curve is constrained to finding minima only. The former quickly finds the local maximum, but the latter has no hope of finding the minimum as it slides off to the left, into the asymptotic abyss.

Truncating at second order in ϵ_k, we have a quadratic approximation to the error, which we add to x_k to get the new iterate x_{k+1},

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}. \tag{4.38}$$

We iterate this update until the absolute change in x_k falls below some tolerance, whereupon the algorithm terminates.

Once more in common with successive parabolic interpolation, this version of the Newton–Raphson method can converge to both minima and maxima. To force convergence to a minimum, we can always move in the decreasing direction by taking the absolute value of second derivative,

$$x_{k+1} = x_k - \frac{f'(x_k)}{|f''(x_k)|}. \tag{4.39}$$

Note that the function is only decreasing within a quadratic approximation, so the update may in fact cause the function value to increase. With this modification, the method may fail to converge if there is a maximum between the starting point and the minimum, as is the case in figure 4.9.

If started close to a stationary point, the Newton-Raphson method for optima usually exhibits a quadratic order of convergence.

4.2.4 Secant Method

If the second derivative of the objective function is not known, it can be approximated by applying the two-point formula to its first derivative; this approach constitutes the secant method for optimisation.

4.2.5 Gradient Descent

Alternatively, we can employ the *gradient-descent method*. In that case, the update is given by

$$x_{k+1} = x_k + \Delta x_k = x_k - \alpha g'_k \tag{4.40}$$

where α is a small, positive, adjustable parameter.

Question 10 The gradient-descent (one dimensional)

Choosing the value of α is a critical consideration: if α is too small, convergence is slow; if α is too large, then we risk jumping back and forth over the minimum, which also results in slow convergence.

1. devise a simple algorithm to choose α;
2. implement the gradient-descent method, using this algorithm.

4.3 Stationary points of functions of many variables

4.3.1 Nelder–Mead Method

Like the golden section method for functions of one variable, the Nelder–Mead method of uses only function values and no derivative information to find a stationary point of a function of many variables.

In one dimension, we use an interval of two points (a, b) to bracket a minimum. We know at least one minimum can be found on the interval because there is a third point c such that $c \in (a, b)$ and $f(a) < f(c) < f(b)$.

To bracket a minimum in two dimensions, we would need to find a region enclosed by a closed curve $\mathcal{C}$, and a probe point d inside the region such that for all points $\forall c \in \mathcal{C}$, $f(d) < f(c)$. In general, this approach is not feasible—in many dimensions, we can't reliably bracket a minimum, and there is no algorithm that is guaranteed to converge to a minimum.

One dimension

Consider once more the one-dimensional problem, minimising $f : \mathbb{R} \mapsto \mathbb{R}$, but this time imagine we cannot bracket a minimum. We begin with two points, (x_1, x_2) where the function values are (f_1, f_2) and $f_1 < f_2$. We say that the point with the lowest function value is the *best* point, and the point with the highest function value is the *worst* point. We want to choose a new point that is better than the worst point. Where should we look?

Assuming that f is decreasing from x_2 to x_1, the new point should be found on the opposite side of x_1 than x_2. We find a new point x_r by **reflecting** the worst point x_2 through the best point x_1 by some amount α. Our first guess is then,

$$x_r = x_1 + \alpha (x_1 - x_2); \quad \alpha > 0. \tag{4.41}$$

If x_r is better than x_1, we can try to **expand** our options by choosing another point,

$$x_e = x_1 + \gamma (x_r - x_1); \quad \gamma > 1. \tag{4.42}$$

We then replace x_2 with whichever of x_r and x_e is better. If, however, x_r is worse than the best point, then we **contract** the worst point toward the best point, choosing a new point,

$$x_c = x_1 + \rho (x_2 - x_1); \quad 0 < \rho \leq 0.5. \tag{4.43}$$

If x_c is better than x_2, we replace x_2 with x_c; otherwise, we **shrink** the interval further toward the best point,

$$x_2 = x_1 + \sigma (x_2 - x_1); \quad 0 < \sigma < 0.5. \tag{4.44}$$

Many dimensions

In n-dimensions, we require $n+1$ points. In one dimension, the points must be different; in two dimensions, the points must not be collinear; in three dimensions, the points must not be coplanar; $i.e.$, for an n-dimensional problem we require a selection of $n + 1$ points that are affinely independent. Such a system of points is called an n-*simplex*.

In n-dimensions, we reflect the worst point through the centroid of the best points,

$$x_o = \sum_{i=1}^{n} \boldsymbol{x}_i. \tag{4.45}$$

The reflected point is thus,

$$\boldsymbol{x_r} = \boldsymbol{x}_o + \alpha (\boldsymbol{x}_o - \boldsymbol{x}_{n+1}). \tag{4.46}$$

If $f_1 \leq f_r < f_n$, we replace $\boldsymbol{x}_{n+1}$ with $\boldsymbol{x}_r$. If $\boldsymbol{x}_r$ is the best point, we expand as before,

$$\boldsymbol{x}_e = \boldsymbol{x}_o + \gamma (\boldsymbol{x}_r - \boldsymbol{x}_o), \tag{4.47}$$

replacing worst point with the best of $\boldsymbol{x}_r$ and $\boldsymbol{x}_e$. Otherwise, $\boldsymbol{x}_r$ is no better than the second-best point, so we contract the worst point toward the centroid,

$$\boldsymbol{x}_c = \boldsymbol{x}_o + \rho (\boldsymbol{x}_{n+1} - \boldsymbol{x}_o). \tag{4.48}$$

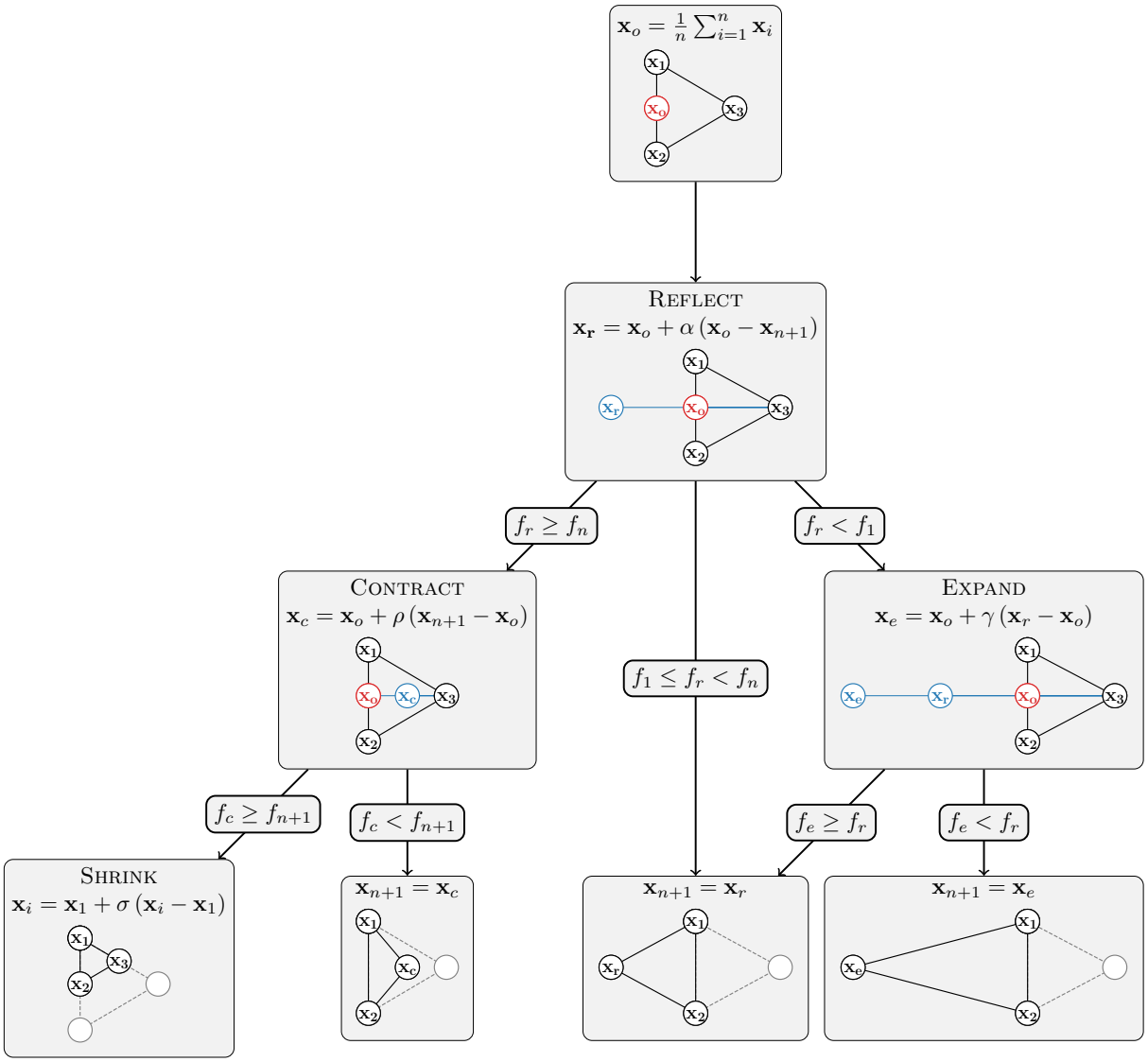

Figure 4.10: Key steps of the Nelder–Mead method, with illustrations demonstrating the two-dimensional problem.

If $\boldsymbol{x}_c$ is better than the worst point, we replace the worst point with $\boldsymbol{x}_c$, and if not, we shrink all the points toward the best point,

$$\boldsymbol{x}_i = \boldsymbol{x}_1 + \sigma\,(\boldsymbol{x}_i - \boldsymbol{x}_1)\,. \tag{4.49}$$

At each iteration, we must relabel the vertices of the simplex so that $f_i < f_{i+1}$. For a the case of a two-dimensional problem, in which the simplex is a triangle, these key steps are illustrated in figure 4.10.

We continue the process until a termination condition is reached, which is when the mean standard deviation of the coordinates $\{\boldsymbol{x}_i\}$ of the simplex is less than some tolerance, or when the standard deviation of the function values $\{f_i\}$ is less than some tolerance.

The method is sensitive to choices of the control parameters, $\{\alpha, \gamma, \sigma, \rho\}$. Standard choices are $\alpha = 1; \gamma = 2; \rho = 0.5; \sigma = 0.5$.

At no point do we bracket a minimum, so convergence is not guaranteed. Even for unimodal functions, if in some region the surface is very flat, the method may converge to a false minimum.

4.3.2 Systems of Linear Equations, Matrix Methods

The remaining methods in this section make use of first and second derivatives of a function to find stationary points. In particular, the Newton method in many dimensions uses the inverse of the matrix of

second derivatives. In order to implement the such methods, we first need to learn some matrix methods. Consider the matrix equation,

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{4.50}$$

which corresponds to the system of equations,

$$\sum_i A_{ji}x_i - b_j = 0. \tag{4.51}$$

The solution is very easy to write down,

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}, \tag{4.52}$$

but less easy to compute, unless $\boldsymbol{A}$ is a diagonal matrix, in which case the inverse $\boldsymbol{A}^{-1}$ is also diagonal,

$$A^{-1}_{ii} = \frac{1}{A_{ii}}. \tag{4.53}$$

We will find it convenient to represent the system of equations as an augmented matrix, so that the equations,

$$\begin{aligned} 2x + y - z &= 8; \\ -3x - y + 2z &= -11; \\ -2x + y + 2z &= -3, \end{aligned} \tag{4.54}$$

are written as,

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right] \tag{4.55}$$

Gauss-Jordan Elimination

Gauss-Jordan elimination is a procedure that transforms $\boldsymbol{A}$ and $\boldsymbol{b}$ so that $\boldsymbol{A}$ is diagonal, but the solution $\boldsymbol{x}$ is unchanged.

We can perform several operations on the equations 4.51 without changing their solution:

- swap the indices of two equations;
- multiply an equation by a non-zero scalar;
- add a multiple of one equation to another.

Translating this to operations on the augmented matrix, we can:

- swap two rows of the matrix;
- multiply a row by a non-zero scalar;
- add a multiple of one row to another.

Take another look at expression 4.55; we can eliminate (zero) the first elements of the second and third rows without changing $\boldsymbol{x}$, by subtracting a multiple of the first row. Namely, we can subtract $3/2$ times the first row from the second row, and -1 times the first row from the third, with the result,

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & 1/2 & 1/2 & 1 \\ 0 & 2 & 1 & 5 \end{array}\right], \tag{4.56}$$

Using the same logic, we can eliminate the second element of the third row by subtracting from it four times the second,

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & 1/2 & 1/2 & 1 \\ 0 & 0 & -1 & 1 \end{array}\right] \tag{4.57}$$

where the left-hand side is a matrix in upper-triangular form. Now our equations read,

$$\begin{aligned} 2x + y - z &= 8; \\ 1/2y + 1/2z &= 1; \\ -z &= 1, \end{aligned} \tag{4.58}$$

i.e., we have solved for z, and by back-substitution can solve for y and then x. Stopping the algorithm here and returning an upper-triangular matrix is known as *Gaussian elimination*.

But why stop there? If we reflect the $\boldsymbol{A}$ in the anti-diagonal and reverse the order of $\boldsymbol{b}$,

$$\left[\begin{array}{ccc|c} -1 & 0 & 0 & 1 \\ 1/2 & 1/2 & 0 & 1 \\ -1 & 1 & 2 & 8 \end{array}\right], \tag{4.59}$$

we *do* change the solution $\boldsymbol{x}$, but only to the extent that the meanings of x and z have been exchanged, *i.e.*,

$$\begin{aligned} -x &= 1; \\ 1/2y + 1/2x &= 1; \\ 2z + y - x &= 8. \end{aligned} \tag{4.60}$$

We can repeat the Gauss–Jordan procedure on this rearranged system until the matrix is diagonal, first eliminating the first elements of the second and third rows,

$$\left[\begin{array}{ccc|c} -1 & 0 & 0 & 1 \\ 0 & 1/2 & 0 & 3/2 \\ 0 & 1 & 2 & 7 \end{array}\right], \tag{4.61}$$

and then eliminating the second element of the third row, At this point, we must remember that we have exchanged x and z, so that our final augmented matrix is,

$$\left[\begin{array}{ccc|c} 2 & 0 & 0 & 4 \\ 0 & 1/2 & 0 & 3/2 \\ 0 & 0 & -1 & 1 \end{array}\right]. \tag{4.62}$$

With $\boldsymbol{A}$ in a diagonal form, the system of equations is full solved,

$$\begin{aligned} 2x = 4 &\implies x = 2; \\ 1/2y = 3/2 &\implies y = 3; \\ -z = 1 &\implies z = -1, \end{aligned} \tag{4.63}$$

or, in augmented matrix form,

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right], \tag{4.64}$$

i.e. $\boldsymbol{Ax} = \boldsymbol{b}; \boldsymbol{A} = \boldsymbol{I} \implies \boldsymbol{x} = \boldsymbol{b}$.

Question 11 Pivoting in Gauss–Jordan elimination

If the first element of the first row of matrix $\boldsymbol{A}$ is zero, you're gonna have a bad time—you can't eliminate the first element of the other rows by subtracting a multiple of the first row. Suggest a solution to this problem.

Determinants

The determinant of a matrix is invariant to the transformations listed in 4.3.2, except that swapping two rows or columns changes its sign. In the Gauss–Jordan procedure outlined above, we didn't swap any rows or columns, so the determinants of the intermediate upper-triangular, and final diagonal matrices are the same as the original matrix. The determinant of an upper-triangular matrix is simply the product of its diagonal elements, so we can determine the determinant of matrix $\boldsymbol{A}$ by performing Gaussian elimination with any non-zero $\boldsymbol{b}$. For example, the augmented matrix,

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 1 \\ -3 & -1 & 2 & 1 \\ -2 & 1 & 2 & 1 \end{array}\right] \tag{4.65}$$

reduces to,

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 1 \\ 0 & 1/2 & 1/2 & 5/2 \\ 0 & 0 & -1 & -8 \end{array}\right], \tag{4.66}$$

so the determinant is $2 \times 1/2 \times -1 = -1$.

Inverses

Inverses are also easy to calculate. First consider the equation,

$$\boldsymbol{A}^{-1}\boldsymbol{b} = \boldsymbol{x}. \tag{4.67}$$

If we set $\boldsymbol{b} = [1, 0, \ldots, 0]$, what is the value of $\boldsymbol{x}$? Precisely the first column of $\boldsymbol{A}^{-1}$! Likewise, choosing $\boldsymbol{b} = \boldsymbol{b}^i$ such that $b^i_j = \delta_{ij}$ finds the i-th column of $\boldsymbol{A}^{-1}$.

We can once more write these equations in the form of an augmented matrix,

$$\left[\begin{array}{ccc|ccc} 2 & 1 & -1 & 1 & 0 & 0 \\ -3 & -1 & 2 & 0 & 1 & 0 \\ -2 & 1 & 2 & 0 & 0 & 1 \end{array}\right], \tag{4.68}$$

which reduces to,

$$\left[\begin{array}{ccc|ccc} -1 & 0 & 0 & -5 & -4 & 1 \\ 0 & \frac{1}{2} & 0 & -1 & -1 & \frac{1}{2} \\ 0 & 0 & 2 & 8 & 6 & -2 \end{array}\right], \tag{4.69}$$

and finally, dividing through by the diagonal,

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 4 & 3 & -1 \\ 0 & 1 & 0 & -2 & -2 & 1 \\ 0 & 0 & 1 & 5 & 4 & -1 \end{array}\right], \tag{4.70}$$

where the right-hand matrix is the inverse of the original matrix $\boldsymbol{A}$.

4.3.3 The Newton–Raphson method (revisited—again)

Our one-dimensional Newton–Raphson approach can also be modified for the multivariable problem. The exact solution, solution at iteration k, and error at iteration k are now given by the vectors $\boldsymbol{\xi}$, $\boldsymbol{x_k}$, and $\boldsymbol{\epsilon_k}$, respectively. Returning to Taylor series,

$$\begin{aligned} f(\xi) = f(\boldsymbol{x_k} + \boldsymbol{\epsilon_k}) &= f(\boldsymbol{x_k}) + \nabla f(\boldsymbol{x_k})^T \cdot \boldsymbol{\epsilon_k} + \mathcal{O}\left(|\boldsymbol{\epsilon_k}^2|\right) \\ \nabla f(\boldsymbol{\xi}) = \boldsymbol{0} &= \nabla f(\boldsymbol{x_k}) + \nabla^2 f(\boldsymbol{x_k}) \cdot \boldsymbol{\epsilon_k} + \mathcal{O}\left(|\boldsymbol{\epsilon_k}^2|\right) \\ \nabla^2 f(\boldsymbol{x_k}) \cdot \boldsymbol{\epsilon_k} &= -\nabla f(\boldsymbol{x_k}) + \mathcal{O}\left(|\boldsymbol{\epsilon_k}^2|\right) \end{aligned} \tag{4.71}$$

Truncating at second order in the error, we have a matrix equation of the form,

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{4.72}$$

where,

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{\epsilon}_k; \\ \boldsymbol{b} &= -\nabla f\left(\boldsymbol{x_k}\right); \\ \boldsymbol{A} &= \nabla^2 f\left(\boldsymbol{x_k}\right). \end{aligned} \tag{4.73}$$

Fortunately, we just spent several pages discussing such problems, so we know that the Newton–Raphson update can be written as,

$$\boldsymbol{\epsilon_k} = -\nabla^2 f\left(\boldsymbol{x_k}\right)^{-1} \cdot \nabla f\left(\boldsymbol{x_k}\right) + \mathcal{O}\left(\left|\boldsymbol{\epsilon}_{\boldsymbol{k}}^2\right|\right), \tag{4.74}$$

where the inverse of the second-derivative matrix can be determined by Gauss–Jordan elimination.

As in the one variable case, the Newton–Raphson method converges q-quadratically to a local stationary point if the starting coordinates are close enough.

Also as before, Newton–Raphson can converge to stationary points of any order. Recall that in the one-dimensional case, the Newton-Raphson method is equivalent to finding the parabola that at the point $\boldsymbol{x}_k$ has the same gradient and curvature as the function we want to minimise. If the curvature is positive, we move in a direction that the function decreases, and if negative, in a direction that the function increases. In the n-dimensional case, the curvature is not a scalar but a matrix, and we are fitting not a parabola but a *quadratic form*, a scalar function of a vector with the form,

$$f\left(\boldsymbol{x}\right) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} + c, \tag{4.75}$$

$\boldsymbol{A}$ symmetric, which can be thought of as a system of orthogonal parabolas.

The search direction is guaranteed to be one in which the function decreases if the parabolas all have positive curvature, which is to say that the matrix of second derivatives is positive-definite, *i.e.*,

$$\forall \boldsymbol{x} \neq \boldsymbol{0}, \quad \boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} > 0 \tag{4.76}$$

(if, instead, $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} >= 0$, the matrix is positive If $\boldsymbol{A}$ is $n \times n$ and symmetric, then it has n eigenvalues and n eigenvectors, $\{(\lambda_i, \boldsymbol{v}_i)\}$. A general vector $\boldsymbol{x}$ can be expressed as a linear combination of these eigenvectors,

$$\boldsymbol{x} = \sum_i^n c_i \boldsymbol{v}_i, \tag{4.77}$$

in which terms the product above is written,

$$\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = \sum_i^n \lambda_i c_i^2. \tag{4.78}$$

For unit vectors $\hat{\boldsymbol{x}}$; $\|\hat{\boldsymbol{x}}\| = 1$, the lowest value expression 4.78 can take is the lowest eigenvalue of $\boldsymbol{A}$, λ_{min}.

It turns out we can construct a positive-definite matrix $\boldsymbol{B}$ with the same eigenvectors as $\boldsymbol{A}$ by adding to it a multiple of the identity matrix,

$$\boldsymbol{B} = \boldsymbol{A} + \mu\boldsymbol{I}, \tag{4.79}$$

for $\mu + \lambda_{\text{min}} > 0$. In the context of the Newton-Raphson method, transforming the matrix of second derivatives in this way guarantees that we move in the direction of decreasing function value.

4.3.4 The gradient-descent method (revisited)

We can easily generalise the gradient-descent update in 4.40 to the multivariable case:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}_k = \boldsymbol{x}_{k+1} - \alpha\frac{\nabla f\left(\boldsymbol{x}_k\right)}{\left|\nabla f\left(\boldsymbol{x}_k\right)\right|} \tag{4.80}$$

The scalar α is typically chosen *via* a *line search method*—a search along the direction opposite to the gradient vector, $\boldsymbol{p}$, for a suitable reduction in the objective function value, *i.e.*, find α such that $f\left(\boldsymbol{x} + \alpha\boldsymbol{p}\right)$ is below some tolerance. One such method is the *backtracking* line search, which is based on the *Armijo-Goldstein condition*.

Starting with a search direction $\boldsymbol{p}$, we find $m = \boldsymbol{p}^T\nabla f\left(\boldsymbol{x}\right)$ (it is required that $m < 0$). Based on a control parameter $c \in (0, 1)$, a step $\alpha\boldsymbol{p}$ is judged to result in an acceptable reduction in the function value if $f\left(\boldsymbol{x} + \alpha\boldsymbol{p}\right) \leq f\left(\boldsymbol{x}\right) + \alpha cm$. If the step fails this test, it is scaled by a second control parameter $\tau \in (0, 1)$ until the test passes. It is a 'backtracking' line search because we start with a relatively large value of α and iteratively reduce it, move back to the original point $\boldsymbol{x}$, until the Armijo-Goldstein condition is satisfied.

Question 12 Backtracking line search

In the sample gradient-descent code, a function 'line_search' is called.

1. implement the backtracking line search.

```
def line_search(f, x, p, gx, c=0.5, t=0.5, max_alpha=1):
    """
    Performs a backtracking line search based on the Armijo-Goldstein condition

    :param f: the objective function
    :type f: callable
    :param x: the point from which to start the line search
    :type x: numpy.ndarray
    :param p: the search direction, unit vector
    :type p: numpy.ndarray
    :param gx: the gradient vector at x
    :type gx: numpy.ndarray
    :param c: scales the threshold above which the step is rejected
    :type c: float
    :param t: scales the step upon rejection
    :type t: float
    :param max_alpha: maximum allowed value of alpha
    :type max_alpha: float
    :return: alpha, the size of a good step to take in the direction p
    :rtype: float
```

Successful convergence of the gradient-descent method to the minimum of the two-dimensional *Rosenbrock function* is illustrated in figure 4.11. The Rosenbrock function has a global minimum in a long, narrow, parabolic-shaped flat valley; converging to this global minimum is difficult, and this function is often used as a performance test problem for optimisation algorithms.

4.3.5 The Broyden–Fletcher–Goldfarb–Shanno (BFGS) method

Computing the Hessian matrix at every step becomes a bottleneck for large systems. A number of methods exist which calculate the Hessian infrequently (or never), and rely on approximate updates to the Hessian during intervening steps.

One popular method of this kind is the *Broyden–Fletcher–Goldfarb–Shanno method* (BFGS). It is not unlike a multivariable version of the one-dimensional secant method. We start with the multivariable secant equation,

$$\nabla^2 f(\boldsymbol{x}_{k+1})(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) \approx \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k) \tag{4.81}$$

For brevity, we will use the notation $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$, $\boldsymbol{y}_k = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$, $\boldsymbol{B}_k = \nabla^2 f(\boldsymbol{x}_k)$. From here, we require an update that preserves both the symmetry and the positive definiteness of the matrix. Notice that using the expression in equation 4.81 does not necessarily preserve either.

One solution is to write the update as,

$$\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + \alpha \boldsymbol{u}\boldsymbol{u}^{\boldsymbol{T}} + \beta \boldsymbol{v}\boldsymbol{v}^{\boldsymbol{T}} \tag{4.82}$$

where $\alpha \boldsymbol{u}\boldsymbol{u}^{\boldsymbol{T}} + \beta \boldsymbol{v}\boldsymbol{v}^{\boldsymbol{T}}$, since they are outer products of a vector with itself, are necessarily both positive semi-definite and symmetric, with their sum a rank-two matrix (unless $\boldsymbol{u}$, $\boldsymbol{v}$ happen to be parallel).

Since we require,

$$\boldsymbol{B}_{k+1}\boldsymbol{s}_k = \boldsymbol{y}_k, \tag{4.83}$$

if choose $\boldsymbol{u} = \boldsymbol{y}_k$ and $\boldsymbol{v} = \boldsymbol{B}_k \boldsymbol{s}_{\boldsymbol{k}}$ we get,

$$\alpha = \frac{1}{\boldsymbol{y}_k^T \boldsymbol{s}_k}; \quad \beta = \frac{1}{\boldsymbol{s}_k^T \boldsymbol{B}_k \boldsymbol{s}_k}, \tag{4.84}$$

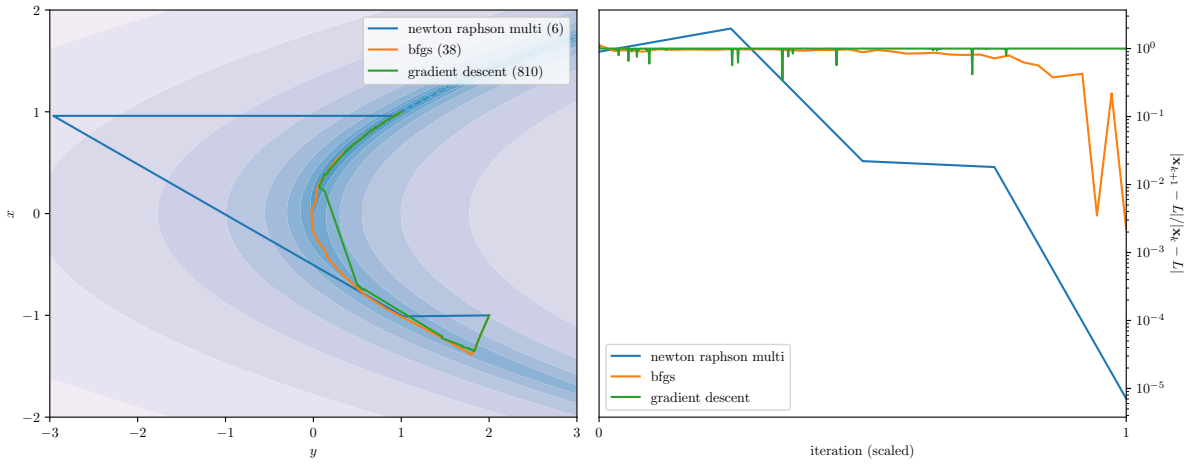

Figure 4.11: (left) Application of the gradient-descent method, the Newton–Raphson method, and the BFGS method to the Rosenbrock function in two dimensions, beginning at the point $(-1, 2)$, which converged to a tolerance in the absolute value of the gradient in 1379, 7, and 38 iterations, respectively. Notice that, in the Newton–Raphson method, the function value does not necessarily decrease at each iteration. (right) The relative change in the difference between the current estimate of the minimum coordinates and the solution, L, for successive iterations. The slowly-converging gradient descent method converges linearly to the solution, whereas the Newton–Raphson calculation converges quadratically.

and the update to the coordinates becomes,

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha \boldsymbol{p} = \boldsymbol{x}_k - \boldsymbol{B}_k^{-1} \nabla \boldsymbol{g}_k, \tag{4.85}$$

and update $\boldsymbol{B}$ as

$$\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + \frac{\boldsymbol{y}_k \boldsymbol{y}_k^T}{\boldsymbol{y}_k^T \boldsymbol{s}_k} - \frac{\boldsymbol{B}_k \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{B}_k^T}{\boldsymbol{s}_k^T \boldsymbol{B}_k \boldsymbol{s}_k}. \tag{4.86}$$

However, since we only need the inverse of $\boldsymbol{B}$, we can rewrite the update using the Sherman–Morrison formula,

$$\boldsymbol{B}_{k+1}^{-1} = \left(I - \frac{s_k y_k^T}{y_k^T s_k} \right) \boldsymbol{B}_k^{-1} \left(I - \frac{y_k s_k^T}{y_k^T s_k} \right) + \frac{s_k s_k^T}{y_k^T s_k} \tag{4.87}$$

As $\boldsymbol{B}_k^{-1}$ is symmetric, and both $\boldsymbol{y}_k^T \boldsymbol{B}_k^{-1} \boldsymbol{y}_k$ and $\boldsymbol{s}_k^T \boldsymbol{y}_k$ are scalars, we can write

$$\boldsymbol{B}_{k+1}^{-1} = \boldsymbol{B}_k^{-1} + \frac{(\boldsymbol{s}_k^T \boldsymbol{y}_k + \boldsymbol{y}_k^T \boldsymbol{B}_k^{-1} \boldsymbol{y}_k)(\boldsymbol{s}_k \boldsymbol{s}_k^T)}{(\boldsymbol{s}_k^T \boldsymbol{y}_k)^2} - \frac{\boldsymbol{B}_k^{-1} \boldsymbol{y}_k \boldsymbol{s}_k^T + \boldsymbol{s}_k \boldsymbol{y}_k^T \boldsymbol{B}_k^{-1}}{\boldsymbol{s}_k^T \boldsymbol{y}_k} \tag{4.88}$$

4.4 First-order saddles

In §4.3 we discussed methods for finding extrema of functions. The gradient descent and BFGS methods will always converge to an extremum, but the Newton–Raphson method can converge to any kind of stationary point.

To understand why, we need to consider the Newton–Raphson update in more detail.

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{H}\left(\boldsymbol{x}_{\boldsymbol{k}}\right)^{-1} \cdot \nabla g\left(\boldsymbol{x}_k\right) \tag{4.89}$$

We first consider a change of coordinates to a coordinate system based on the Hessian eigenvectors,

$$\boldsymbol{B}^{-1} \boldsymbol{H} \boldsymbol{B} = \boldsymbol{\Lambda} \tag{4.90}$$

with $\boldsymbol{B}$ and $\boldsymbol{\Lambda}$ the matrix of Hessian eigenvectors and (diagonal) matrix of eigenvalues, respectively. The matrix $\boldsymbol{B}$ allows to transform to new orthogonal coordinates $\boldsymbol{w} = \boldsymbol{B}^{-1} \boldsymbol{x}$. The Newton-Raphson update

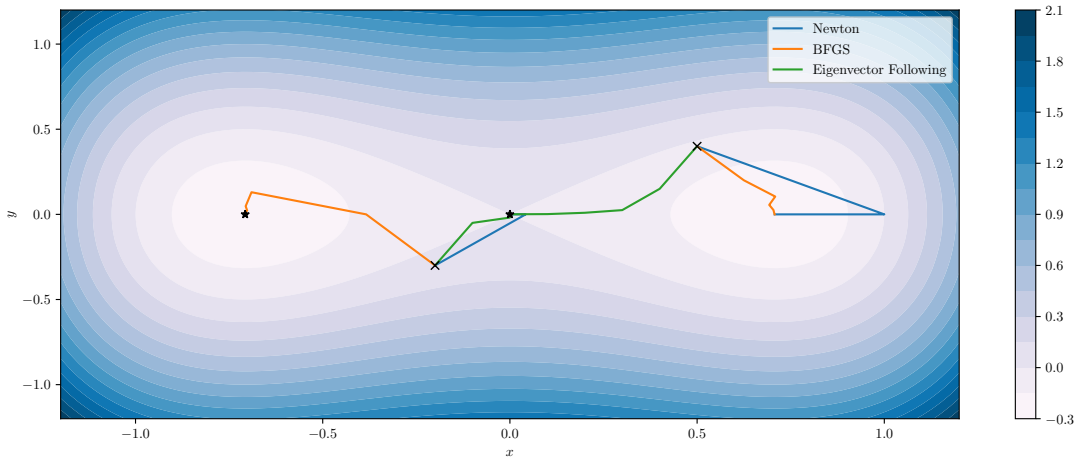

Figure 4.12: Comparison of the Newton–Raphson, BFGS, and eigenvector-following methods in finding first-order saddles. The plotted function is $f(x,y) = x^4 - x^2 + y^2$, which has degenerate minima at $(x,y) = \left(1/\sqrt{(2)}, 0\right), \left(1/\sqrt{(2)}, 0\right)$, and an index 1 saddle at $(x,y) = (0,0)$, indicated by black stars.

and predicted function value change then become

$$\begin{aligned} x_{k+1} &= -\boldsymbol{\Lambda}^{-1} \nabla g\left(\boldsymbol{w}_k\right) \\ \Delta g &= -\frac{1}{2} \nabla g\left(\boldsymbol{w}_k\right) \cdot \boldsymbol{\Lambda}^{-1} \nabla g\left(\boldsymbol{w}_k\right) \end{aligned} \tag{4.91}$$

A step is taken in each eigendirection, where the step in direction i decreases (increases) the function value if $\lambda_i > 0$ ($\lambda_i < 0$). As a result, whether the Newton–Raphson converges to an extremum or some other stationary point depends on the eigenspectrum of the Hessian matrix (see figure 4.12 for an example of this phenomenon). During the course of an optimisation, the number of negative Hessian eigenvalues may change, so, to converge to an index 1 saddle, it is not sufficient to begin a Newton–Raphson search from a point where the Hessian has one negative eigenvalue. However, given that we have found such a point, there are methods that allow us to do so.

4.4.1 Eigenvector-following

A method for finding index 1 saddles, the essence of *eigenvector-following* is to maximise in one direction, while minimising in all others.

Say we are at a point in space where the Hessian has one negative eigenvalue. The first thing to do is identify the associated eigenvector. Many algorithms exist to diagonalise matrices. With this eigenvector in hand, we want to move a small amount along it in the direction that the function value is increasing, using, for example, a line search. From the new configuration, we want to minimise in all directions orthogonal to that eigenvector. To do so, we project that eigenvector out of the gradient vector,

$$\nabla f' = \nabla f - \left(\nabla f^T \boldsymbol{v}_{\min}\right) \boldsymbol{v}_{\min} \tag{4.92}$$

and proceed to minimise in the direction opposite to $\nabla f'$ using for example, the BFGS method. We repeat these steps until the root-mean-square (rms) gradients both along the eigendirection and in the orthogonal subspace fall below some tolerance.

4.4.2 The Rayleigh-Ritz ratio

For large systems, inverting the Hessian to find its eigenvectors can become a bottleneck. To avoid this problem, we can make use of the *Rayleigh–Ritz ratio*,

$$\lambda\left(\boldsymbol{v}\right) = \frac{\boldsymbol{v}^T \boldsymbol{H}\left(\boldsymbol{x}\right) \boldsymbol{v}}{\boldsymbol{v}^T \boldsymbol{v}} \tag{4.93}$$

$$\nabla\lambda\left(\boldsymbol{v}\right)=\frac{\left(\boldsymbol{v}^{T}\boldsymbol{v}\right)\left(\boldsymbol{H}\boldsymbol{v}+\boldsymbol{H}^{T}\boldsymbol{v}\right)-2\left(\boldsymbol{v}^{T}\boldsymbol{H}\boldsymbol{v}\right)\boldsymbol{v}}{\left(\boldsymbol{v}^{T}\boldsymbol{v}\right)^{2}} \tag{4.94}$$

which is a stationary point when $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{H}$, with eigenvalue $\lambda\left(\boldsymbol{v}\right)$, and a unique minimum when $\boldsymbol{v}=\boldsymbol{v}_{\min}$ is the eigenvector associated with the smallest eigenvalue of $\boldsymbol{H}$, $\lambda_{\min}$. Of course, if the Hessian is not known at all, we can approximate it by applying the central difference method to the gradient.

$\boldsymbol{v}_{\min}$ and $\lambda_{\min}$ can be found by gradient descent or the BFGS method.

Question 13 The Rayleigh-Ritz method

1. Implement the Rayleigh-Ritz method.

```
def rayleigh_ritz(h, x, tol=1e-3):
    """
    Finds the smallest eigenvalue and associated eigenvector
    of the Hessian matrix at point x by minimising the Rayleigh-Ritz ratio,

        λ(v) = (v.T . H . v) / (v.T . v)

    :param h: function to get the Hessian
    :type h: callable
    :param x: point at which to evaluate the Hessian
    :type x: numpy.ndarray
    :param tol: convergence tolerance for the gradient of λ
    :type tol: float
    :return: the smallest eigenvalue and associated (normalised) eigenvector
    :rtype: (float, numpy.ndarray,)

    """
```

Chapter 5

Global Optimisation

So far we have studied the methods of Monte Carlo and molecular dynamics as means of investigating the structure, thermodynamics, and dynamics of a system. Central to both approaches in the potential energy surface (PES): in Monte Carlo simulations, the probability of accepting a move from one state to another is determined by their relative potential energies; in molecular dynamics, the time evolution of the system is determined by forces on the interacting bodies, which are in part determined by the underlying PES.

We have also studied various methods for finding local optima of functions. If we knew the energies of all the local minima on a PES, along with the volumes of configuration space associated with those minima, we would be able to compute any thermodynamic property of the system it described.

At low temperatures, a system does not have enough energy to freely explore the PES; consequently, its properties are adequately described by the properties of a few low-energy minima. At lower temperatures still, at equilibrium, the properties of the system are those of the lowest-energy minimum, the *global minimum*.

Global optimisation is the field dedicated to discovering the global optimum of a system, be it the structure of the native state of a protein, the short strand of DNA that binds most strongly to a viral genome, or perhaps the lowest-energy packing of irregular particles in space. By no means is its scope restricted to solving problems in the physical sciences; maximising the profit of a business, the performance of a microchip, or the structural integrity of a building are all motivators for developing global optimisation techniques.

Global optimisation strategies fall into three main categories:

1. **deterministic methods**, where a solution is certain to be found, *e.g.*, linear programming problems;
2. **stochastic methods**, which rely on random variables, *e.g.*, Monte Carlo sampling
3. and **heuristic methods**, which also rely on random numbers, but attempt to search the search space in a more or less intelligent way, *e.g.*, tabu search, ant colony optimisation, evolutionary algorithms...

Global optimisation on a multidimensional PES is a non-deterministic polynomial-time (NP) hard problem; [Wille and Vennik, 1985, Ngo and Marks, 1992] no known algorithm guarantees that the optimal solution will be found on a timescale proportional to some power of the system size. The first category, then, is not useful here; recourse to either stochastic or heuristic approaches is required.

Heuristic methods can improve upon stochastic methods by introducing extra assumptions, or by sacrificing flexibility (there is no such thing as a free lunch). That is to say, while heuristic methods often offer superior performance for some subset of problems, they need not and typically do not offer superior performance in optimisation problems generally.

In this chapter, we will look at two conceptually different global optimisation techniques: basin-hopping, a stochastic technique widely applied in the physical sciences to find ground-state structures; and genetic (or evolutionary) algorithms, a class of heuristic methods which draw inspiration from natural selection and genetics.

5.1 Basin-Hopping

A popular stochastic method is basin-hopping (BH), [Li and Scheraga, 1987, Wales and Doye, 1997] in which a transformed PES is explored by Monte Carlo (MC) steps. Each point on the PES is mapped to the

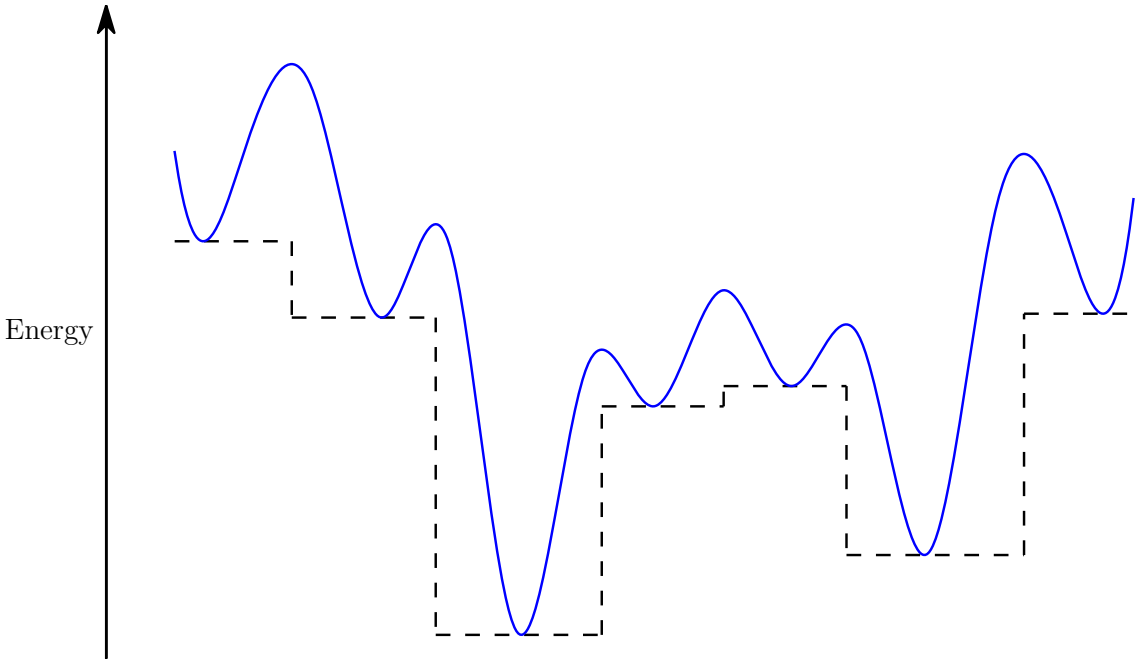

Figure 5.1: The effect of the basin transformation described by equation 5.1 on a one-dimensional potential function. The solid blue line and dashed black lines represent the original and transformed potentials, V and $\tilde{V}$, respectively.

local minimum reached by an approximate steepest-descent path originating at that point,

$$\tilde{V}(\mathbf{r}) = V(\tilde{\mathbf{r}}) = \min\{V(\mathbf{r})\}, \tag{5.1}$$

where $\min\{V(\mathbf{r})\}$ is defined as the energy at the local minimum with coordinates $\tilde{\mathbf{r}}$, obtained from a local minimisation of the potential energy in coordinate space (i.e. a quench), starting from $\mathbf{r}$. Thus the surface is partitioned into disjoint sets of 'basins of attraction', as shown in figure 5.1, each containing a local minimum and all points that quench to it. The 'energy' of the basin of attraction is defined as the energy of the local minimum it contains. Exploration of the transformed surface is less impeded by energetic barriers compared to traditional MC or molecular dynamics simulations.

The basic BH scheme proceeds as follows:

1. a perturbation is applied to the coordinates of the old configuration, $\mathbf{r}_o$;
2. the perturbed coordinates, $\mathbf{r}'_o$, are quenched to a new configuration with coordinates $\mathbf{r}_n$;
3. the new configuration is accepted with probability

$$p(o \to n) = \min\left(1, e^{-\beta \Delta V}\right) \tag{5.2}$$

   where $\Delta V = V_n - V_o$, V_o and V_n are the energies at $\mathbf{r}_o$ and $\mathbf{r}_n$, and $\beta = 1/kT$, where k is Boltzmann's constant and T is the 'temperature' parameter.
4. the next step begins from the new configuration if accepted, and the old one otherwise.

Steps 1, 3, and 4 should remind you of the basic Monte Carlo algorithm. However, in step 1, we have more freedom to choose the perturbation than in Monte Carlo, since, if we are only interested in the global minimum, we **do not have to satisfy detailed balance**. Additionally, the 'temperature' no longer has a physical meaning; it is just a control parameter like any other in the algorithms discussed in chapter 4.

5.1.1 Choosing Basin-Hopping Parameters

In its simplest form, BH has only two parameters: the 'temperature' used in the acceptance step, and the perturbation applied. The perturbation could be a random displacement in translation or orientational

coordinates, in which case the magnitude of the displacement is of prime importance. We can think of the succession of configurations as a trajectory on the transformed surface, $\tilde{V}$: if the magnitude of the displacement is too small, the trajectory will never leave the original basin of attraction; too large, and the quality of the exploration of the landscape will degenerate to a random search. The temperature parameter controls how likely it is that an energy-increasing configuration will be accepted into the trajectory. Once again, if it is too small, the trajectory may never go anywhere; too large, and the trajectory will oversample high energy regions of the PES, making it inefficient.

Rather than choose a fixed magnitude for the translational steps, or a fixed BH temperature, we can allow both to vary according to separate criteria: the step ratio, S_{rat}, and the temperature ratio, T_{rat}. [Farrell and Wales, 2014] S_{rat} is defined as the probability that a perturbation/minimisation step ends in a different basin to the one in which it begins,

$$S_{\mathrm{rat}} = p(\boldsymbol{X}_n \ncong \boldsymbol{X}_o), \tag{5.3}$$

where X_o and X_n are the Cartesian coordinates before and after the step, respectively. Two configurations correspond to the same basin if the Euclidean distance between them, minimised with respect to rotation, translation, and permutation of identical particles, is less than 10^{-3}. The minimised distance can be determined by, for example, using the stochastic shortest augmenting path algorithm. [Wales and Carr, 2012]

T_{rat} is defined as the probability that the result of a perturbation/minimisation step is accepted into the BH trajectory,

$$T_{\mathrm{rat}} = p(o \to n | \boldsymbol{X}_n \ncong \boldsymbol{X}_i), \tag{5.4}$$

which means that A is simply

$$A = (1 - S_{\mathrm{rat}}) + S_{\mathrm{rat}} T_{\mathrm{rat}}, \tag{5.5}$$

i.e., the sum of the probability of remaining in the same minimum, $1 - S_{\mathrm{rat}}$, and the probability of accepting a step that ends in a different basin, $S_{\mathrm{rat}} T_{\mathrm{rat}}$. This decomposition of the usual acceptance ratio can provide a parametrisation of the BH step size and temperature which avoids pathological outcomes and, being bounded between zero and one, is independent of the length and energy scales associated with the system, and so more closely reflects the topography of the landscape.

5.1.2 The Perturbation

Translational Displacements

For systems such as clusters of Lennard-Jones particles, the perturbation can be as simple as randomly displacing some or all of the particle coordinates,

$$\boldsymbol{X}_i^{'} = \boldsymbol{X}_i + \boldsymbol{x}. \tag{5.6}$$

where $\boldsymbol{x}$ is a random displacement. The magnitude of $\boldsymbol{x}$ must be chosen to avoid the pathological behaviours outlined in §5.1.1.

Orientational Displacements

For systems whose constituents possess an orientation, such as particles with a magnetic dipole, ellipsoidal particles, or molecules, orientational displacements may also prove useful,

$$\boldsymbol{P}_i^{'} = \boldsymbol{P}_i + \boldsymbol{p}. \tag{5.7}$$

where $\boldsymbol{p}$ is a random displacement.

Surface Moves

We can try to reduce the energy of a cluster using *surface moves*. The outermost particles in a cluster comprise the cluster surface; the other particles comprise the *core*. A surface move might involve the following steps:

1. find the most weakly-bound particle on the cluster surface (the highest-energy particle);

2. generate a set of random positions on the surface and compute the energy required to move the particle to those positions (the insertion energy);

3. place the particle at the position with the lowest insertion energy

Surface moves are often used in combination with other kinds of moves, *e.g.*, translational displacements, producing a hierarchical scheme,

1. take a large step in configuration space with a translational displacement;

2. minimise;

3. do basin-hopping on the new structure for some number of surface moves;

4. accept or reject the final structure according to the Metropolis criterion.

Symmetry-Based Moves

There is strong evidence to suggest that structures with high symmetry lie at both the low and high extremes of the energy spectrum. [Wales, 1998] While global minima of complex systems may not have exact symmetries, quasi-symmetries can still be found. Oakley *et al.* proposed a global optimisation scheme that chooses moves according to approximate symmetries of a cluster. [Oakley et al., 2013] In short, a symmetrical core is identified, and symmetry positions on the surface generated. Surface particles are then permuted among those symmetry positions. The candidate positions on the surface can also be chosen according to a symmetry or quasi-symmetry of the system. Even though the global minimum of 98 Lennard-Jones particles has only C_s symmetry, this method reduces the time taken to find the global minimum by a factor of 70.

Molecular Dynamics

The perturbation can also be a short molecular dynamics trajectory. This approach is useful for complex molecules such as proteins, polynucleotides (DNA, RNA), and other biomolecules, for which a random displacement may result in physically unreasonable structures, *e.g.*, inversion around a chiral centre. Randomness is introduced by randomly generating the momenta/torques of each component at the beginning of the trajectory.

In Goedecker's 'minima hopping' method, [Goedecker, 2004], an MD step is used. This method differs from the example basin-hopping method outlined above in that it does not use a the Metropolis criterion, but a 'threshold' for the accept/reject part; a step is accepted if the energy increases by some threshold energy E_{diff}, which is adjusted on-the-fly to give an average acceptance probability of 0.5.

Flooding

Flooding is actually a perturbation of the landscape rather than the coordinates. The idea was introduced in the context of rare events in molecular dynamics [Grubmüller, 1995,Voter, 1997], and involves increasing the energy of parts of the landscape which we would like to escape. In the context of global optimisation, flooding is used to avoid or quickly pass through regions of the surface that have already been visited, to avoid getting stuck in traps and improve the survey of the landscape. One application is 'basin-paving,' where basins are filled in (their energy is increased) as they are visited. [Zhan et al., 2006] This action decreases the probability that those basins will be visited again during the simulation. A similar approach is 'temperature basin-paving,' wherein the temperature rather than the energy is increased. [Shanker and Bandyopadhyay, 2011] In this manner, the basins can be revisited, but will quickly be escaped as the local 'temperature' is high.

5.2 Genetic Algorithms

Among a population, those fittest individuals, who most suited to their environments, flourish, coming together to produce offspring which share some of their successful characteristics; those least fit, perish, and without issue. Over time, the population as a whole becomes more well-adapted to their condition. This

process Charles Darwin called "evolution by natural selection", and is nature's answer to the optimisation problem, "which characteristics are those of the individual most-suited to his environment?"

Since Darwin, we have learned that those successful individuals propagate their successful characteristics in the form of chromosomes, pages of the construction manual we call the genome. Those individuals that the inherit from among the best of their parents' chromosomes go on to be even more suited to their environments, and pass on those chromosomes to the next generation, whereas the weakest chromosomes and those individuals that inherit them, pass out of the gene pool, and out the mortal coil, respectively. Occasionally, random changes, "mutations," will occur in those chromosomes, helping or hindering the success of the individuals that possess them.

Genetic algorithms, or evolutionary algorithms, are a class of techniques that employ analogues of competition, mating, and mutation to find the solution that optimises a cost function, *i.e.*, the individual most suited to his environment.

5.2.1 Encoding

Basin-hopping works by mapping each point $\{r_1, r_2, \ldots, r_n\} = \mathbf{r}$ in the solution space to a local minimum, $\tilde{\mathbf{r}}$. Genetic algorithms instead map each point to a binary string, the so-called chromosome. First, each point $\mathbf{r}$ is mapped by a linear transformation to a point $\tilde{\mathbf{r}}$ such that the $\tilde{r}_i \in [0, 1]$. Then, in the same way decimal numbers are encoded as binary numbers in a computer, the $\tilde{r}_i$ are approximated as a sum of reciprocal powers of two,

$$\tilde{r}_i = \sum_{j=1}^{p} \frac{y_{ij}}{2^j} \tag{5.8}$$

where the y_{ij} are either zero or one, and p is an integer that determines the precision of the encoding in bits. The array $y = \{y_{ij} | 1 < j < p; 1 < i < n\}$ is the chromosome, and the sub-array $y_i = \{y_{ij} | 1 < j < p\}$ is called the ith gene of the chromosome. Note that, given a finite choice of p, many configurations r map to the same chromosome y, but every y maps to a unique r.

5.2.2 A Basic Genetic Algorithm

The first step is to **initialise** a fixed number, n_r, of chromosomes to compose the "gene pool" of the first "generation". Subsequent generations are produced by iterating the following three steps:

1. **select** from the gene pool the $n_g = n_r/2$ chromosomes that will survive into the next generation;
2. from the survivors, choose pairs of chromosomes (parents) to **crossover** together to form new chromosomes (offspring);
3. introduce random **mutations** into the chromosomes of some of the population

Selection

The purpose of a genetic algorithm is to minimise a cost function. Selection is done in such a way that the "fittest" chromosomes, those which represent relatively optimal solutions to the cost function, are more likely to proceed to the next generation, and those least fit, more likely to be discarded.

The simplest approach is to select the n_g fittest chromosomes from the gene pool, discarding the remainder. Alternatively, chromosomes can be drawn at random from the gene pool, with a probability weighted by their fitness. A popular approach is that of tournaments, wherein the gene pool is split into pairs, and only the fittest individual in each pair proceeds to the next generation.

Selection, however it takes place, is analogous to the Metropolis step in basin-hopping. Restrictive selection (only the very best chromosomes are chosen) is like basin-hopping with a low temperature, resulting in a search gets trapped in local optima in the solution space. Liberal selection (choosing chromosomes at random) is like basin-hopping with a high temperature, resulting in a random search.

Crossover

Crossover operations are the first of two ways that a genetic algorithm explores the solution space. In a process analogous sexual reproduction, pairs of chromosomes are chosen from the gene pool and mixed

together in some way to produce new, different chromosomes, representing new elements of the solution space. One type of crossover scheme is the "single-point crossover", wherein two chromosomes are cut in half, and two new chromosomes are created by splicing opposite halves together. For example, parents $y_1 = \{01010101\}$ and $y_2 = \{11001100\}$ would produce offspring $w_3 = \{01011100\}$ and $w_4 = \{11000101\}$ if the crossover point was taken to be the centre of the arrays.

As with in the selection step, there are many possibilities for choosing pairs of parents in the crossover step.

Mutation

If using crossover operations alone, over the course of many generations the gene pool can get stuck in a local minimum, comprising chromosomes that represent very similar solutions encompassing a small volume of solution space.

Mutation operations add some extra diversity into the gene pool by randomly flipping bits in some of the chromosomes. The probability of flipping a bit, p_f, is a control parameter, and works similarly to step size parameter in basin hopping: too low, and the search doesn't go anywhere; too high, and the resulting chromosomes will have a random (likely relatively poor) fitness, and will be rejected at the selection stage.

Appendix A

More Programming with Python

Here we introduce some aspects of the python language that we couldn't fit into the main programming course.

A.1 Strings

The `string` data type is used for supplying input to a program, displaying the progress or state of a computation, and conveying error messages and warnings, among other things. Strings are collections of letters, numbers, and other characters, which are sized (they have a length), iterable (the elements are the characters), and subscriptable (they can be indexed like lists). With some exceptions, any combination of ASCII characters put inside single quotes `'` or double quotes `"` is a string literal. The string `printable` in the `string` module contains all of the ASCII characters that are considered printable. In particular, tabs and newlines are represented by `\t` and `\n`.

```
>>> s = 'I am a string'
>>> t = "I'm a string, too"  # notice the ' inside the "..."
>>> s, t
('I am a string', 'I'm a string, too')
>>>
>>> from string import printable
>>> len(printable)
100
>>> printable[:50]
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN'
>>> printable[50:]
'OPQRSTUVWXYZ!"#%$&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
>>>
```

When we try to `print` an object, the `__str__` method of that object (or `__repr__` method, if a `__str__` method is not defined) is called to return a string object. The same thing happens when we pass the object to the `str` function,

```
>>> x = 45
>>> x
45
>>> s_x = str(x)
>>> s_x
'45'
>>> type(s_x)
<class 'str'>
>>>
```

We often want to print to the screen some information about the state of our program; the values assigned to names, the sum of this, and the standard deviation of that. The simplest way to this is to pass a comma-separated list of strings and names to the print function,

```
>>> total = 0
>>> for counter in range(10):
...     total += counter
...
>>> print("the total is", total, "and the counter is", counter)
the total is 45 and the counter is 9
>>>
```

To get more control over the appearance of the output, we can use *string formatting syntax*. In Python, there are three (!) string formatting syntaxes. The first, and oldest, is *%-formatting*. This syntax will be familiar to users of `C`, `awk`, and many other languages. It is **not** recommended. The other two we will now discuss in more detail.

A.1.1 f-Strings

f-Strings, also called *formatted string literals*, are a new in Python 3.6. The syntax is the same as for normal strings, except that the first `'` or `"` is preceded by the letter f (i.e., `f'` or `f"`, and that code appearing between curly braces `{...}` is interpreted as an expression,

```
>>> f'the total is {total} and the counter is {counter}'
'the total is 45 and the counter is 9'
>>> f'{4} times {5} is {4 * 5}'
'4 times 5 is 20'
>>> f'half the total is {total / 2}'
'half the total is 22.5'
>>>
```

Along with the expression itself we can pass a format specifier to give further control over how the result is printed, e.g., choosing between fixed point and exponent notation,

```
>>> from math import pi
>>> pi
3.141592653589793
>>> f'pi = {pi:10.3f}'  # floating point format, 10 chars, 3 chars after the decimal
'pi =      3.142'
>>> f'pi = {pi:10.3e}'  # scientific notation, 10 chars, 3 chars after the decimal
'pi =  3.142e+00'
>>>
```

See the documentation for more details on the format specification mini-language.

A.1.2 str.format()

Formatting *via* the `format` method uses the same format specification mini-language, but substituted expressions are supplied to the `format` method of the string, and the initial `f` is dropped. Substitutions can be made according to the position of an expression in the expression list, or by keyword, and can exploit list or dictionary unpacking,

```
>>> s = 'the total is {} and the counter is {}'
>>> s.format(45, 9)
'the total is 45 and the counter is 9'
>>> s = 'the total is {0} and the counter is {1}'  # by index
>>> s.format(45, 9)
'the total is 45 and the counter is 9'
>>> s = '{0}{1}{2}{1}{2}{1}'  # by index
>>> s.format('b','a','n')
'banana'
>>> s.format(*'ban')  # with list unpacking
'banana'
>>> s = 'the total is {total} and the counter is {counter}'  # by keyword
```

```
>>> s.format(total=45, counter=9)
'the total is 45 and the counter is 9'
>>> s = 'the total is {total:10.5f} and the counter is {counter}'  # with format specifier
>>> s.format(total=45, counter=9)
'the total is   45.00000 and the counter is 9'
>>>
```

Which of these methods is best depends on the situation. Personally, I prefer to use f-strings for short strings, to enhance code readability. For longer strings, or when unpacking is useful, I prefer `str.format`. For example, see the following code for writing an ndarray of coordinates in the standard xyz format:

```
def write_xyz(coordinates):
    """

    Writes the coordinates in array coordinates to a string in the xyz format.

    https://en.wikipedia.org/wiki/XYZ_file_format

    :param coordinates: coordinates of the system
    :type coordinates: ndarray, shape (particles, dimensions)
    :return: xyz string
    :rtype: str
    """
    particles, dimensions = coordinates.shape
    header = f'{particles}\n\n'  # short f-string
    particle_line = "O{:20.10f}{:20.10f}{:20.10f}\n"  # my particles are usually all the same
    xyz = header + particles * particle_line  # string arithmetic

    return xyz.format(*coordinates.flatten())  # unpack coordinates into format method call
```

Strings implement a whole host of methods besides `format` for convenient string manipulation, analysis, localisation... before you implement a string function, check here to make sure you aren't reinventing the wheel.

Of course, to become a true master, a king of strings (or queen of chars), you must understand regular expressions. In Python, regular expressions are implemented in the re module.

A.2 Sets

Sets are sized, but unordered, collections of unique objects. They can be created by passing a sequence of hashable objects (read: immutable objects with a `__hash__` method) to the `set` function. New elements are inserted with the `add` method (not `append`—"append" means "add to the end;" since sets have no well-defined order, we cannot reliably add an object to the end of set). Sets support the same operations as other sequences, as well as set operations you will be familiar with from mathematics.

```
>>> set()  # the empty set
set()
>>> set([1, 2, 3, 4, 3, 2, 1])  # a set initialised from a list
{1, 2, 3, 4}
>>> a = set([1,2,3,4]); b = set([3, 4, 5, 6])
>>> a | b  # union
{1, 2, 3, 4, 5, 6}
>>> a & b  # intersection
{3, 4}
>>> (a - b), (b - a)  # difference, or complement
({1, 2}, {5, 6})
>>> a ^ b  # symmetric difference
{1, 2, 5, 6}
>>>
```

Elements of a sets are guaranteed to be unique. Checking whether an object is in a set is an $\mathcal{O}(1)$ operation (the time taken is independent of the size of the set) compared with $\mathcal{O}(N)$ for a list (the time taken is linear in the length of the list. Set (and list) membership is tested *via* the `in` operator. There are also methods for checking for subsets.

```
>>> a, b, c = 1, set([1, 2]), set([1, 2, 3])
>>> a in b
True
>>> b.issubset(c)
True
>>> c.issuperset(b)
True
>>> b.isdisjoint(c)
False
>>>
```

A.3 Dictionaries

Dictionaries are mapping objects that map hashable values to arbitrary objects. They consist of `key:value` pairs. A dictionary indexed by a key returns the corresponding value; a new value can be bound to a key by assigning the value to the dictionary indexed by the key. The keys are unique, but many keys may be associated with the same value.

```
>>> d = dict([("a", 1), ("b", 2), ("c", 3)])  # dict function
>>> d = {"a":1, "b":2, "c":3}  # dict literal
>>> d.keys()
dict_keys(['a', 'b', 'c'])
>>> d.values()
dict_values([1, 2, 3])
>>> d.items()
dict_items([('a', 1), ('b', 2), ('c', 3)])
>>> "a" in d, 1 in d
(True, False)
>>> d["a"], d["b"], d["c"]
(1, 2, 3)
>>> d["e"] = 5
>>> d
{'a': 1, 'b': 2, 'c': 3, 'e': 5}
>>> d["a"] = 6
>>> d
{'a': 6, 'b': 2, 'c': 3, 'e': 5}
>>>
```

A.4 Comprehensions and Generators

Comprehensions provide a convenient way of abbreviating simple loops to create lists, dictionaries, sets, and generators. Readers familiar with set-builder notation will immediately recognise the value this syntax. For example, in set-builder notation, the set of integers less than n can be written as

$$\{x \in \mathbb{Z}^+ | x < n\}$$

This can be implemented in Python as a while loop,

```
>>> numbers = []
>>>
>>> n = 5
>>> numbers = []
```

```
>>> i = 1
>>> while i < 5:
...     numbers.append(i)
...     i += 1
...
>>> numbers
[1, 2, 3, 4]
```

or, more succinctly, as a for loop using the range function,

```
>>> n = 5
>>> numbers = []
>>> for i in range(1, n+1):
...     numbers.append(i)
...
>>> numbers
[1, 2, 3, 4, 5]
```

With a *list comprehension*, this list can be created in one line,

```
>>> n = 5
>>> numbers = [i for i in range(1, n + 1)]
>>> numbers
[1, 2, 3, 4, 5]
```

To obtain a set instead of a list, simply replace the `[...]` with `{...}` to obtain a *set comprehension*,

```
>>> n = 5
>>> numbers = {i for i in range(1, n + 1)}
>>> numbers
{1, 2, 3, 4, 5}
```

For a generator, the `[...]` must be replaced with `(...)`, returning a *generator expression*,

```
>>> n = 5
>>> numbers = (i for i in range(1, n + 1))
>>> numbers
<generator object <genexpr> at 0x???????????>
```

Generators are evaluated *lazily*—the elements are only created when needed. The next object in the generator expression can be returned by calling the `next` function with the generator expression as its argument. This can save memory compared to list comprehensions,

```
>>> n = 10**100
>>> list_of_numbers = [i for i in range(1, n + 1)]  # MemoryError ! do not execute!
>>> generator_of_numbers = (i for i in range(1, n + 1))  # no problem
>>> next(generator_of_numbers)
1
>>> next(generator_of_numbers)
2
>>> for i in generator_of_numbers:
...     print(i)
...
3
4
# etc
```

Unlike lists obtained from list comprehensions, generators obtained from generator expressions (indeed, generators of all kinds) cannot be indexed,

```
>>> generator_of_numbers[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable
```

Comprehensions can be equipped with if clauses,

```
>>> [i for i in range(10) if i % 3 == 0]
[0, 3, 6, 9]
>>> [i for i in range(20) if i % 3 == 0 or i % 5 == 0]
[0, 3, 5, 6, 9, 10, 12, 15, 18]
>>>
```

More complex generators can are best defined with definition statements,

```
>>> def positive_integers():  # yields every positive integer
...     i = 0
...     while True:
...             i += 1
...             yield i
...     return
...
>>> genexp = positive_integers()
>>> next(genexp)
1
>>> next(genexp)
2
>>> next(genexp)
3
>>>
```

where the elements of the generator are return *via* `yield` statements. Generators defined in this fashion are reusable and can be documented clearly. They can of course form part of comprehensions and generator expressions,

```
>>> square_integers = (i*i for i in positive_integers())
>>> next(square_integers)
1
>>> next(square_integers)
4
>>> next(square_integers)
9
>>>
```

Here is an example of a generator that can be used to generate (potentially infinitely many) prime numbers:

```
from math import inf


def integers(minimum=1, maximum=inf):
    """

    Yields integers n, minimum <= n <= maximum

    :param minimum: the smallest integer
    :type minimum: int
    :param maximum: the largest integer (default math.inf, i.e. infinite generation)
    :type maximum: int
    :return: generator for integers
    :rtype: Iterator[:class:`int`]
```

```
    """
    i = minimum
    while i <= maximum:
        yield i
        i += 1


def trial_division(maximum=inf):
    """

    Generates primes, p up to maximum using trial division.

    :param maximum: maximum possible value of p
    :type maximum: int
    :return: generator of primes
    :rtype: Iterator[:class:`int`]
    """
    primes = [2, 3, 5]
    for p in primes:
        yield p
    for i in integers(7, maximum):
        maxp = int(i ** 0.5)
        for p in primes:
            if p > maxp:
                primes.append(i)
                yield i
                break
            elif i % p == 0:
                break
```

A.5 Anonymous Functions

`lambda` functions are convenient for writing one-time-only functions that will be passed as arguments to other functions. Consider the `pair_potential` code from an earlier section. We passed as arguments the coordinates of the particles, the potential function, and a tuple of arguments that will be passed to the potential function. We can rewrite the function call

```
>>> pair_potential(xs, potential=ljpotential, potential_args=(1.0, 1.0)
```

as

```
>>> pair_potential(xs, potential=lambda x: ljpotential(x, 1.0, 1.0))
```

Here, the expression `lambda x: ljpotential(x, epsilon=1.0, sigma=1.0)` evaluates to a function of a single variable, namely, `ljpotential` with the `epsilon` and `sigma` arguments both fixed to 1.0. However, if you print this object, you won't learn anything useful about it,

```
>>> from simulations import lj_potential
>>> anon = lambda x: lj_potential(x, epsilon=1.0, sigma=1.0)
>>> anon
<function <lambda> at 0x7f21c7554e50>
```

hence the term 'anonymous function.' Anonymity can make it difficult to trace errors. A less flexible, but more informative alternative is provided by the `partial` function in the `functools` module,

```
>>> import functools
>>> named_func = functools.partial(lj_potential, epsilon=1.0, sigma=1.0)
```

```
>>> named_func
functools.partial(<function lj_potential at 0x7f21c7554ee0>, epsilon=1.0, sigma=1.0)
```

A.6 Classes and Dataclasses

A deep discussion of python classes is well outside the scope of this course, and would constitute a long and unnecessary diversion. Here they are only introduced, along with the very useful `dataclass` decorator.

In this course, we have discussed many types, among them, numeric types, such as `int` and `float`, and container types, such as `list` and `tuple`.

Just as we can define our own functions using the `def` keyword, we can define our own types, or *classes*, by using the `class` keyword. A classic example is a 2D vector class,

```
1  class Vector2D:
2      """
3      A 2D-vector class defining vector addition
4
5      >>> u = Vector2D(1,2)
6      >>> v = Vector2D(2,3)
7      >>> print(u)
8      Vector2D(x=1, y=2)
9
10     >>> repr(u)
11     'Vector2D(x=1, y=2)'
12
13     >>> u + v
14     Vector2D(x=3, y=5)
15
16     """
17     def __init__(self, x: float, y: float):
18         self.x = x
19         self.y = y
```

An instance of a class is created when the class is called like a function,

```
>>> from classes.vector2D import Vector2D
>>> u = Vector2D(x=1.0, y=2.0)
```

where the RHS is an expression which evaluates to the return value of the `__init__` function, defined on lines 17–19 with arguments `x=1.0, y=2.0`. If we try to print the object, we see something like the following,

```
>>> print(u)
<classes.vector2D.Vector2D object at 0x7f8ea5739df0>
```

showing the class name and memory address of the instance. We can *override* this behaviour by defining the `__str__` and `__repr__` methods of `Vector2D`. Methods with double underscores on either side of their names are called special methods, and they determine how an object behaves under certain circumstances. When the `str` function is called with an object as its argument, the return value is actually the return value of the `__str__` method of that object. Similarly, a call `repr(instance)` evaluates to `instance.__repr__()`. If the `__str__` method is not defined, `str(object)` also returns `instance.__repr__()`.

Let's define a `__repr__` function,

```
21     def __repr__(self) -> str:
22         return f'{self.__class__.__name__}(x={self.x}, y={self.y})'
```

Now printing the instance gives a little more information,

```
>>> print(u)
Vector2D(x=1, y=2)
```

Can we add two vectors together?

```
>>> u = Vector2D(x=1.0, y=2.0)
>>> v = Vector2D(x=1.0, y=2.0)
>>> u + v
Traceback (most recent call last):
  File "/home/compphys/anaconda3/envs/computational_physics/lib/python3.9/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vector2D' and 'Vector2D'
```

Not yet! Look at the error: it's a `TypeError`, because the `+` operator doesn't know what to do when either or both of its arguments are type `Vector2D`. The syntax `x + y` is actually shorthand for `x.__add__(y)` — another special method—so we can define addition between `Vector2D`s by defining the `__add__` method,

```
24      def __add__(self, other: 'Vector2D') -> 'Vector2D':
25          return Vector2D(self.x + other.x, self.y + other.y)
```

This time around we ought not to get a `TypeError`,

```
>>> u = Vector2D(x=1.0, y=2.0)
>>> v = Vector2D(x=1.0, y=2.0)
>>> u + v
Vector2D(x=2.0, y=4.0)
```

Here is the full class definition for ease of reference:

```
1   class Vector2D:
2       """
3       A 2D-vector class defining vector addition
4
5       >>> u = Vector2D(1,2)
6       >>> v = Vector2D(2,3)
7       >>> print(u)
8       Vector2D(x=1, y=2)
9
10      >>> repr(u)
11      'Vector2D(x=1, y=2)'
12
13      >>> u + v
14      Vector2D(x=3, y=5)
15
16      """
17      def __init__(self, x: float, y: float):
18          self.x = x
19          self.y = y
20
21      def __repr__(self) -> str:
22          return f'{self.__class__.__name__}(x={self.x}, y={self.y})'
23
24      def __add__(self, other: 'Vector2D') -> 'Vector2D':
25          return Vector2D(self.x + other.x, self.y + other.y)
```

Addition between instances of a user-defined class has no one-size fits all implementation. It may even be unreasonable to define addition, *e.g.*, in the case of addition of `dict`s,

```
{1: 'a'} + {2: 'b'}
Traceback (most recent call last):
  File "/home/compphys/anaconda3/envs/computational_physics/lib/python3.9/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Such methods have to be implemented on a case-by-case basis. If, however, the new class can be thought of as essentially a tuple with some methods attached to it, the `__init__`, `__str__` and `__repr__` methods, along with many others, do have sensible defaults, and can be generated automatically using the `dataclass` decorator.

```
from dataclasses import dataclass


@dataclass
class Vector2D:
    """
    A 2D-vector class defining vector addition.

    >>> u = Vector2D(1,2)
    >>> v = Vector2D(2,3)
    >>> print(u)
    Vector2D(x=1, y=2)

    >>> repr(u)
    'Vector2D(x=1, y=2)'

    >>> u + v
    Vector2D(x=3, y=5)

    """
    x: float
    y: float

    def __add__(self, other: 'Vector2D') -> 'Vector2D':
        return Vector2D(self.x + other.x, self.y + other.y)
```

In this way we can dispense with much 'boilerplate' code (code that has to be written the same way every time a new class is defined), and spend more time working on interesting problems.

Appendix B

Useful Packages for Scientific Programming and Data Analysis

Here we introduce a few python packages that are widely used in scientific programming (descriptions taken in part or whole from their respective websites).

B.1 NumPy

Numerical **Py**thon. NumPy is a high-performance library upon which many other mathematical and scientific packages are based. Introduced in §1.4.1.

B.2 Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures ina variety of hardcopy formats and interactive environments across platforms. It tries to make easy things easy and hard things possible. Introduced in section §1.4.2.

B.3 SciPy (library)

Scientific **Py**thon. The SciPy library is one of the core packages that make up the SciPy stack. It provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimisation.

B.4 SymPy

Symbolic **Py**thon. SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible.

B.5 Pandas

The Python Data Analysis Library. pandas is an open source, BSD-licensed library providing easy-to-use, high-performance data structures and data analysis tools for the Python programming language.

B.6 NetworkX

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

B.7 scikit-learn

scikit-learn provides a platform for performing machine learning in python. It contains simple and efficient tools for data mining and data analysis and is built on NumPy, SciPy, and matplotlib.

B.8 pele

Python **e**nergy **l**andscape **e**xplorer. pele is a package of tools for calculations involving optimisation and exploration on energy landscapes. The core routines are broken into two parts: Basinhopping, for finding the global minimum of an energy landscape, and for building up databases of minima, and DoubleEndedConnect, for finding minimum energy paths on the energy landscape between two minima. *Not widely-used, but a favourite of mine.—JDF*

Appendix C

More NumPy

C.1 Vectorise?

Standard operators are vectorised. So are `numpy` functions. In general, others functions, such as `math` functions, are not vectorised. `math` functions will try to turn their array arguments into scalars. Only `numpy` arrays with shapes `(n0, n1, ..., nk)` where $n_i = 1 \; \forall i$ can be converted to scalars. If the argument cannot be converted to a Python scalar, a `TypeError` is raised.

Such functions can made to work with `minpy` arrays *via* the `numpy.vectorize` function. The result of `numpy.vectorize` is the same as writing a loop over the array values—it is a *convenience function*—there are no performance gains.

```
def fsquared(x, f):
    """
    return f(x)**2

    >>> import math, numpy
    >>> from numpy import array
    >>> x, xs, xss = 1, array([1]), array([0, 1, 2])
    >>> fsquared(x, math.exp)  #doctest: +ELLIPSIS
    7.389056...
    >>> fsquared(x, numpy.exp)  #doctest: +ELLIPSIS
    7.389056...
    >>> fsquared(xs, math.exp)  #doctest: +ELLIPSIS
    7.389056...
    >>> fsquared(xs, numpy.exp)  #doctest: +ELLIPSIS
    array([7.389056...])
    >>> fsquared(xss, math.exp)  #doctest: +ELLIPSIS
    Traceback (most recent call last):
    TypeError: only size-1 arrays can be converted to Python scalars
    >>> fsquared(xss, numpy.vectorize(math.exp))  #doctest: +ELLIPSIS
    array([ 1.        ,  7.389056... , 54.598150...])
    >>> fsquared(xss, numpy.exp)  #doctest: +ELLIPSIS
    array([ 1.        ,  7.389056... , 54.598150...])

    :param x: number(s)
    :type x: numeric
    :param f: function
    :type f: callable
    :return: f(x)**2
    :rtype: type(x)
    """

    return f(x)**2
```

C.2 Einstein Summation

`einsum` is easily my favourite `numpy` function. Take the example of finding the interia tensor of a system of particles. r are the coordinates, $i, j, \ldots$ are particle indices, and $\alpha, \beta, \ldots$ are dimensional indices. Then, the inertia tensor element $I_{\alpha\beta}$ is given by:

$$I_{\alpha\beta} = e_{\alpha\gamma\epsilon} e_{\beta\delta\epsilon} K_{\gamma\delta} \tag{C.1}$$

where

$$K_{\alpha\beta} = m_i r_{\alpha i} r_{\beta i} \tag{C.2}$$

and $e_{\alpha\beta\gamma}$ is a Levi-Civita symbol,

$$\begin{aligned} e_{xyz} = e_{yzx} = e_{zxy} &= 1 \\ e_{xzy} = e_{yxz} = e_{zyx} &= -1 \\ e_{\alpha\beta\gamma} &= 0 \text{ if any pair indices are identical} \end{aligned} \tag{C.3}$$

`einsum` allows us to implement these equations very easily; we simply provide a string listing the relevant indices along with the relevant arrays, and `numpy` takes care of the rest:

```
import numpy as np


def levi_civita(dtype=float):
    from numpy import zeros
    e_tensor = zeros([3, 3, 3], dtype=dtype)
    for i in [(0, 1, 2), (1, 2, 0), (2, 0, 1)]:
        e_tensor.itemset(i, 1)
        e_tensor.itemset(i[::-1], -1)
    e_tensor.setflags(write=False)
    return e_tensor


LEVI_CIVITA = levi_civita(int)


def centre_of_mass(pos, masses):
    return np.einsum('i, ia', masses, pos) / masses.sum()


def get_inertia_tensor(pos, masses):
    pos0 = pos - centre_of_mass(pos, masses)
    k = np.einsum('i, ia, ib -> ab', masses, pos0, pos0)
    return np.einsum('age, bde, gd -> ab', LEVI_CIVITA, LEVI_CIVITA, k)
```

Neat, huh?

Bibliography

[Farrell and Wales, 2014] Farrell, J. D. and Wales, D. J. (2014). Clusters of coarse-grained water molecules. *J. Phys. Chem. A* , 118(35):7338–7348.

[Goedecker, 2004] Goedecker, S. (2004). Minima hopping: An efficient search method for the global minimum of the potential energy surface of complex molecular systems. *J. Chem. Phys.* , 120(21):9911–9917.

[Grubmüller, 1995] Grubmüller, H. (1995). Predicting slow structural transitions in macromolecular systems: Conformational flooding. *Phys. Rev. E* , 52(3):2893.

[Li and Scheraga, 1987] Li, Z. and Scheraga, H. A. (1987). Monte carlo-minimization approach to the multiple-minima problem in protein folding. *Proc. Natl. Acad. Sci. USA* , 84(19):6611–6615.

[Ngo and Marks, 1992] Ngo, J. T. and Marks, J. (1992). Computational complexity of a problem in molecular structure prediction. *Protein Eng.* , 5(4):313–321.

[Oakley et al., 2013] Oakley, M. T., Johnston, R. L., and Wales, D. J. (2013). Symmetrisation schemes for global optimisation of atomic clusters. *Phys. Chem. Chem. Phys.* , 15(11):3965–3976.

[Shanker and Bandyopadhyay, 2011] Shanker, S. and Bandyopadhyay, P. (2011). Monte carlo temperature basin paving with effective fragment potential: An efficient and fast method for finding low-energy structures of water clusters (h2o) 20 and (h2o) 25. *J. Phys. Chem. A* , 115(42):11866–11875.

[Voter, 1997] Voter, A. F. (1997). A method for accelerating the molecular dynamics simulation of infrequent events. *The Journal of chemical physics*, 106(11):4665–4677.

[Wales, 1998] Wales, D. J. (1998). Symmetry, near-symmetry and energetics. *Chem. Phys. Lett.* , 285(5-6):330–336.

[Wales and Carr, 2012] Wales, D. J. and Carr, J. M. (2012). Quasi-continuous interpolation scheme for pathways between distant configurations. *J. Chem. Theory Comput.* , 8(12):5020–5034.

[Wales and Doye, 1997] Wales, D. J. and Doye, J. P. K. (1997). Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *J. Phys. Chem. A* , 101(28):5111–5116.

[Wille and Vennik, 1985] Wille, L. T. and Vennik, J. (1985). Computational complexity of the ground-state determination of atomic clusters. *J. Phys. A* , 18(8):L419.

[Zhan et al., 2006] Zhan, L., Chen, J. Z., and Liu, W.-K. (2006). Monte carlo basin paving: an improved global optimization method. *Physical Review E*, 73(1):015701.