First Semester in Numerical Analysis with Python

Yaning Liu
Department of Mathematical and Statistical Sciences
University of Colorado Denver
Denver CO 80204

Giray Ökten
Department of Mathematics
Florida State University
Tallahassee FL 32306

To my wife Yanjun Pan-YL

Contents

Preface

The book is based on "First semester in Numerical Analysis with Julia", written by Giray Ökten[1]. The contents of the original book are retained, while all the algorithms are implemented in Python (Version 3.8.0). Python is an open source (under OSI), interpreted, general-purpose programming language that has a large number of users around the world. Python is ranked the third in August 2020 by the TIOBE programming community index[2], a measure of popularity of programming languages, and is the top-ranked interpreted language. We hope this book will better serve readers who are interested in a first course in Numerical Analysis, but are more familiar with Python for the implementation of the algorithms.

The first chapter of the book has a self-contained tutorial for Python, including how to set up the computer environment. Anaconda, the open-source individual edition, is recommended for an easy installation of Python and effortless management of Python packages, and the Jupyter environment, a web-based interactive development environment for Python as well as many other programming languages, was used throughout the book and is recommended to the readers for easy code developement, graph visualization and reproducibility.

The book was also inspired by a series of Open Educational Resources workshops at University of Colorado Denver and supported partially by the professional development funding thereof. Yaning Liu also thanks his students in his Numerical Analysis classes, who enjoyed using Python to implement the algorithms and motivated him to write a Numerical Analysis textbook with codes in Python.

Yaning Liu
August 2020
Denver, Colorado

Giray Ökten
August 2020
Tallahassee, Florida

[1] https://open.umn.edu/opentextbooks/textbooks/first-semester-in-numerical-analysis-with-julia
[2] https://www.tiobe.com/tiobe-index/

Chapter 1

Introduction

1.1 Review of Calculus

There are several concepts and facts from Calculus that we need in Numerical Analysis. In this section we will list some definitions and theorems that will be needed later. For the most part functions in this book refer to real valued functions defined on real numbers $\mathbf{R}$, or an interval $(a, b) \subset \mathbf{R}$.

Definition 1. 1. A function f has the limit L at x_0, written as $\lim_{x \to x_0} f(x) = L$, if for any $\epsilon > 0$, there exists $\delta > 0$ such that $|f(x) - L| < \epsilon$ whenever $0 < |x - x_0| < \delta$.

2. A function f is continuous at x_0 if $\lim_{x \to x_0} f(x) = f(x_0)$, and f is continuous on a set A if it is continuous at each $x_0 \in A$.

3. Let $\{x_n\}_{n=1}^{\infty}$ be an infinite sequence of real numbers. The sequence has the limit x, i.e., $\lim_{n \to \infty} x_n = x$ (or, written as $x_n \to x$ as $n \to \infty$) if for any $\epsilon > 0$, there exists an integer $N > 0$ such that $|x_n - x| < \epsilon$ whenever $n > N$.

Theorem 2. *The following are equivalent for a real valued function f :*

1. f is continuous at x_0

2. If $\{x_n\}_{n=1}^{\infty}$ is any sequence converging to x_0, then $\lim_{n \to \infty} f(x_n) = f(x_0)$.

Definition 3. We say $f(x)$ is differentiable at x_0 if

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

exists.

Notation: $C^n(A)$ denotes the set of all functions f such that f and its first n derivatives are continuous on A. If f is only continuous on A, then we write $f \in C^0(A)$. $C^\infty(A)$ consists of functions that have derivatives of all orders, for example, $f(x) = \sin x$ or $f(x) = e^x$.

The following well-known theorems of Calculus will often be used in the remainder of the book.

Theorem 4 (Mean value theorem). *If $f \in C^0[a,b]$ and f is differentiable on (a,b), then there exists $c \in (a,b)$ such that $f'(c) = \frac{f(b)-f(a)}{b-a}$.*

Theorem 5 (Extreme value theorem). *If $f \in C^0[a,b]$ then the function attains a minimum and maximum value over $[a,b]$. If f is differentiable on (a,b), then the extreme values occur either at the endpoints a, b or where f' is zero.*

Theorem 6 (Intermediate value theorem). *If $f \in C^0[a,b]$ and K is any number between $f(a)$ and $f(b)$, then there exists $c \in (a,b)$ with $f(c) = K$.*

Theorem 7 (Taylor's theorem). *Suppose $f \in C^n[a,b]$ and $f^{(n+1)}$ exists on (a,b), and $x_0 \in (a,b)$. Then, for $x \in (a,b)$*

$$f(x) = P_n(x) + R_n(x)$$

where P_n is the nth order Taylor polynomial

$$P_n(x) = f(x_0) + f'(x_0)(x - x_0) + f''(x_0)\frac{(x-x_0)^2}{2!} + \ldots + f^{(n)}(x_0)\frac{(x-x_0)^n}{n!}$$

and R_n is the remainder term

$$R_n(x) = f^{(n+1)}(\xi)\frac{(x-x_0)^{n+1}}{(n+1)!}$$

for some ξ between x and x_0.

Example 8. Let $f(x) = x\cos x - x$.

1. Find $P_3(x)$ about $x_0 = \pi/2$ and use it to approximate $f(0.8)$.
2. Compute the exact value for $f(0.8)$, and the error $|f(0.8) - P_3(0.8)|$.
3. Use the remainder term $R_3(x)$ to find an upper bound for the error $|f(0.8) - P_3(0.8)|$. Compare the upper bound with the actual error found in part 2.

Solution. 1. First note that $f(\pi/2) = -\pi/2$. Differentiating f we get:

$$f'(x) = \cos x - x\sin x - 1 \Rightarrow f'(\pi/2) = -\pi/2 - 1$$
$$f''(x) = -2\sin x - x\cos x \Rightarrow f''(\pi/2) = -2$$
$$f'''(x) = -3\cos x + x\sin x \Rightarrow f'''(\pi/2) = \pi/2.$$

Therefore

$$P_3(x) = -\pi/2 - (\pi/2+1)(x-\pi/2) - (x-\pi/2)^2 + \frac{\pi}{12}(x-\pi/2)^3.$$

Then we approximate $f(0.8)$ by $P_3(0.8) = -0.3033$ (using 4-digits with rounding).

2. The exact value is $f(0.8) = -0.2426$ and the absolute error is $|f(0.8) - P_3(0.8)| = 0.06062$.

3. To find an upper bound for the error, write

$$|f(0.8) - P_3(0.8)| = |R_3(0.8)|$$

where

$$R_3(0.8) = f^{(4)}(\xi)\frac{(0.8-\pi/2)^4}{4!}$$

and ξ is between 0.8 and $\pi/2$. We need to differentiate f one more time: $f^{(4)}(x) = 4\sin x + x\cos x$. Since $0.8 < \xi < \pi/2$, we can find an upper bound for $f^{(4)}(x)$, and thus an upper bound for $R_3(0.8)$, using triangle inequality:

$$|R_3(0.8)| = \left| f^{(4)}(\xi)\frac{(0.8-\pi/2)^4}{4!} \right| = |4\sin\xi + \xi\cos\xi|\,(0.01471)$$
$$\leq 0.05883|\sin\xi| + 0.01471|\xi||\cos\xi|.$$

Note that on $0.8 < \xi < \pi/2$, $\sin\xi$ is a positive increasing function, and $|\sin(\xi)| < \sin(\pi/2) = 1$. For the second term, we can find an upper bound by observing $|\xi|$ attains a maximum value of $\pi/2$ on $0.8 < \xi < \pi/2$, and $\cos\xi$, which is a decreasing positive function on $0.8 < \xi < \pi/2$, has a maximum value of $\cos(0.8) = 0.6967$. Putting these together, we get

$$|R_3(0.8)| < 0.05883(1) + (0.01471)(\pi/2)(0.6967) \approx 0.07493.$$

Therefore, our estimate for the actual error (which is 0.06062 from part 2) is 0.07493.

Exercise 1.1-1: Find the second order Taylor polynomial for $f(x) = e^x \sin x$ about $x_0 = 0$.

a) Compute $P_2(0.4)$ to approximate $f(0.4)$. Use the remainder term $R_2(0.4)$ to find an upper bound for the error $|P_2(0.4) - f(0.4)|$. Compare the upper bound with the actual error.

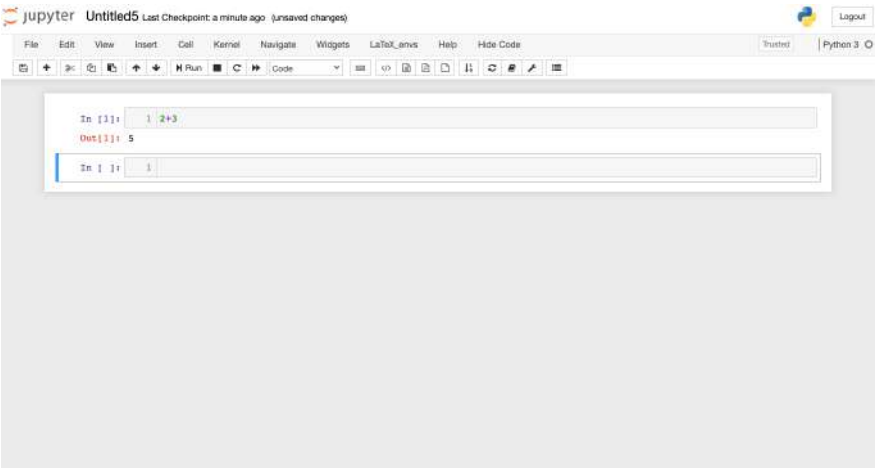
b) Compute $\int_0^1 P_2(x)dx$ to approximate $\int_0^1 f(x)dx$. Find an upper bound for the error using $\int_0^1 R_2(x)dx$, and compare it to the actual error.

1.2 Python basics

We recommend using the free and open-source Python package manager Anaconda (Individual Edition https://www.anaconda.com/products/individual). Go to the webpage and choose the installer according to your operating system. At the time of writing this book, Python Version 3.8 is installed by default, which is also the version we use in this book. There are different environments and editors to run Python. Here we will use the Jupyter environment, which will be ready after Anaconda is installed. There are several tutorials and other resources on Python at https://www.python.org/ where one can find up-to-date information on Python.

The Jupyter environment uses the so-called Jupyter notebook where one can write and edit a Python code, run the code, and export the work into various file formats including Latex and pdf. Most of our interaction with Python will be through the Jupyter notebooks. Note that Anaconda, once installed, has already preinstalled a large number of commonly used Python packages, so it is not necessary to install new packages for the purpose of running the codes in this book.

After installing Anaconda, open a Jupyter notebook by following Anaconda → Jupyter Notebook → new → Python 3 . Here is a screenshot of my notebook:



Let's start with some basic computations.

```
In [1]: 2+3
```

```
Out[1]: 5
```

Now we import the **sin** and **log** function, as well as the π constant from the **math** package,

```
In [2]: from math import sin, pi, log
```

and compute $\sin(\pi/4)$:

```
In [3]: sin(pi/4)
```

```
Out[3]: 0.7071067811865475
```

One way to learn about a function is to search for it online in the Python documentation https://docs.python.org/3/. For example, the syntax for the logarithm function in the **math** package is $\log(x[, b])$ where b is the optional base. If b is not provided, the natural logarithm of x (to base e) is computed.

```
In [4]: log(4, 2)
```

```
Out[4]: 2.0
```

NumPy arrays

NumPy is a useful Python package for array data structure, random number generation, linear algebra algorithms, and so on. A NumPy array is a data structure that can be used to represent vectors and matrices, for which the computations are also made easier. Import the NumPy package and define an alias (np) for it.

```
In [5]: import numpy as np
```

Here is the basic syntax to create a 1D NumPy array (representing a vector):

```
In [6]: x = np.array([10, 20, 30])
        x

Out[6]: array([10, 20, 30])
```

The following line of code shows the entries of the created array are integers of 64 bits.

```
In [7]: x.dtype

Out[7]: dtype('int64')
```

If we input a real, Python will change the type accordingly:

```
In [8]: x = np.array([10, 20, 30, 0.1])
        x

Out[8]: array([10. , 20. , 30. ,  0.1])

In [9]: x.dtype

Out[9]: dtype('float64')
```

A 1D NumPy array does not assume a particular row or column arrangement of the data, and hence taking transpose for a 1D NumPy array is not valid. Here is another way to construct a 1D array, and some array operations:

```
In [10]: x = np.array([10*i for i in range(1, 6)])
         x

Out[10]: array([10, 20, 30, 40, 50])

In [11]: x[-1]
```

```
Out[11]: 50

In [12]: min(x)

Out[12]: 10

In [13]: np.sum(x)

Out[13]: 150

In [14]: x = np.append(x, 99)
         x

Out[14]: array([10, 20, 30, 40, 50, 99])

In [15]: x[3]

Out[15]: 40

In [16]: x.size

Out[16]: 6
```

The NumPy package has a wide range of mathematical functions such as sin, log, etc., which can be applied elementwise to an array:

```
In [17]: x = np.array([1,2,3])

In [18]: np.sin(x)

Out[18]: array([0.84147098, 0.90929743, 0.14112001])
```

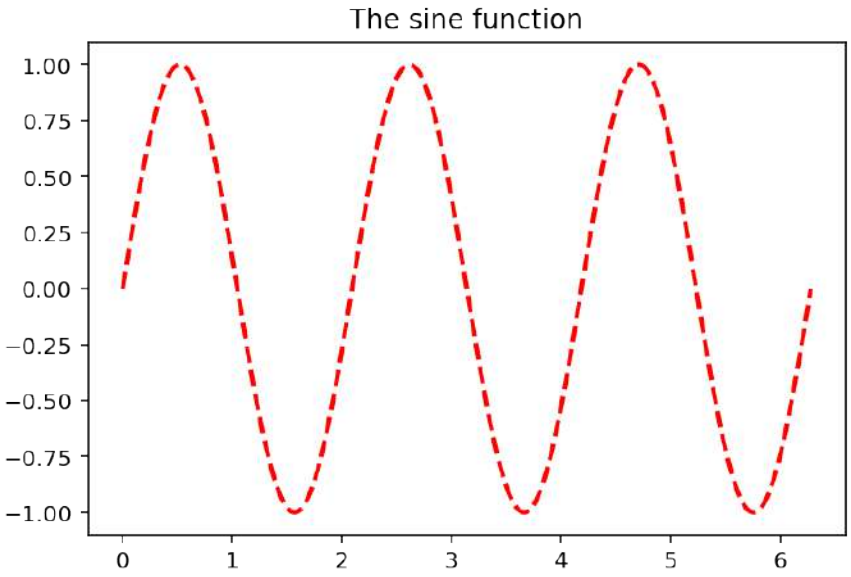
Plotting

There are several packages for plotting functions and we will use the PyPlot package. The package is preinstalled in Anaconda. To start the package, use

```
In [19]: import matplotlib.pyplot as plt
         %matplotlib inline
```

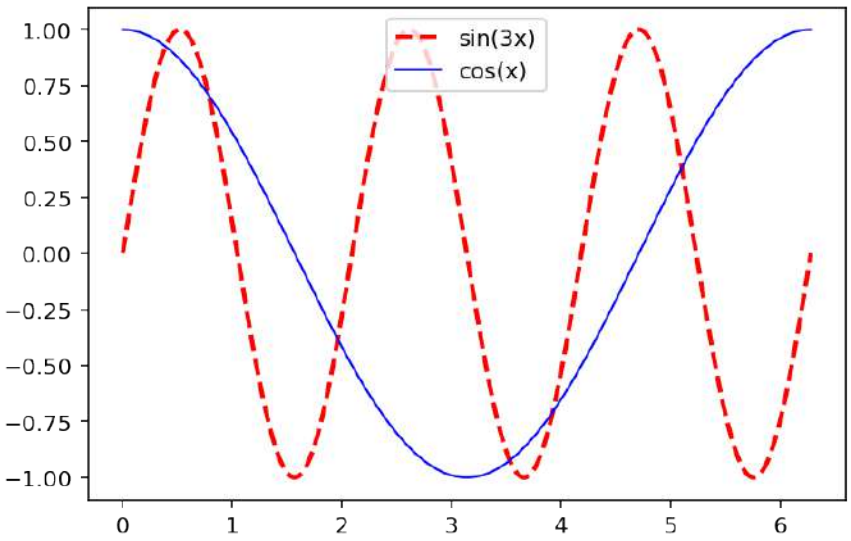
The following piece of code is a simple example of plotting with PyPlot.

```
In [20]: x = np.linspace(0, 2*np.pi, 1000)
         y = np.sin(3*x)
         plt.plot(x, y, color='red', linewidth=2.0, linestyle='--')
         plt.title('The sine function');
```



Let's plot two functions, $\sin 3x$ and $\cos x$, and label them appropriately.

```
In [21]: x = np.linspace(0, 2*np.pi, 1000)
         y = np.sin(3*x)
         z = np.cos(x)
         plt.plot(x, y, color='red', linewidth=2.0, linestyle='--', label='sin(3x)')
         plt.plot(x, z, color='blue', linewidth=1.0, linestyle='-', label='cos(x)')
         plt.legend(loc='upper center');
```



Matrix operations

NumPy uses 2D arrays to represent matrices. Let's create a 3×3 matrix (2D array):

```
In [22]: A = np.array([[-1, 0.26, 0.74], [0.09, -1, 0.26], [1,1,1]])
         A

Out[22]: array([[-1.  ,  0.26,  0.74],
                [ 0.09, -1.  ,  0.26],
                [ 1.  ,  1.  ,  1.  ]])
```

Transpose of A is computed as:

```
In [23]: A.T

Out[23]: array([[-1.  ,  0.09,  1.  ],
                [ 0.26, -1.  ,  1.  ],
                [ 0.74,  0.26,  1.  ]])
```

Here is its inverse.

```
In [24]: np.linalg.inv(A)
```

```
Out[24]: array([[-0.59693007,  0.22740193,  0.38260375],
                [ 0.08053818, -0.82433201,  0.15472807],
                [ 0.51639189,  0.59693007,  0.46266818]])
```

To compute the product of A and the inverse of A, use:

```
In [25]: np.dot(A, np.linalg.inv(A))
```

```
Out[25]: array([[ 1.00000000e+00,  1.24617643e-17, -1.74319952e-17],
                [-3.98224397e-17,  1.00000000e+00,  1.74319952e-17],
                [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00]])
```

Let's try matrix vector multiplication. Define some vector v as:

```
In [26]: v = np.array([0,0,1])
         v
```

```
Out[26]: array([0, 0, 1])
```

Now try np.dot(A, v) to multiply them.

```
In [27]: np.dot(A, v)
```

```
Out[27]: array([0.74, 0.26, 1.  ])
```

To solve the matrix equation $Ax = v$, type:

```
In [28]: np.linalg.solve(A, v)
```

```
Out[28]: array([0.38260375, 0.15472807, 0.46266818])
```

The solution to $Ax = v$ can be also computed as $x = A^{-1}v$ as follows:

```
In [29]: np.dot(np.linalg.inv(A), v)
```

```
Out[29]: array([0.38260375, 0.15472807, 0.46266818])
```

Powers of A can be computed as:

```
In [30]: np.linalg.matrix_power(A, 5)
```

```
Out[30]: array([[-2.80229724,  0.39583437,  2.76651959],
                [ 0.12572728, -1.39301724,  0.84621677],
                [ 3.74250756,  3.24338756,  4.51654028]])
```

Logic operations

Here are some basic logic operations:

```
In [31]: 2 == 3

Out[31]: False

In [32]: 2 <= 3

Out[32]: True

In [33]: (2==2) or (1<0)

Out[33]: True

In [34]: (2==2) and (1<0)

Out[34]: False

In [35]: (4 % 2) == 0 # Check if 4 is an even number

Out[35]: True

In [36]: (5 % 2) == 0

Out[36]: False

In [37]: (5 % 2) == 1 # Check if 5 is an odd number

Out[37]: True
```

Defining functions

There are two ways to define a function. Here is the basic syntax:

```
In [38]: def squareit(x):
             return x**2

In [39]: squareit(3)

Out[39]: 9
```

There is also a compact form to define a function, if the body of the function is a short, simple expression:

```
In [40]: cubeit = lambda x: x**3

In [41]: cubeit(5)

Out[41]: 125
```

Suppose we want to pick the elements of an array that are greater than 0. This can be done using:

```
In [42]: x = np.array([-2,3,4,5,-3,0])
         x[x>0]

Out[42]: array([3, 4, 5])
```

To count the number of elements that are greater than 0 in the array above, use

```
In [43]: x[x>0].size

Out[43]: 3
```

Types

In Python, there are several types for integers and floating-point numbers such as int8, int64, float32, float64, and more advanced types for Boolean variables and strings. When we write a function, we do not have to declare the type of its variables: Python figures what the correct type is when the code is compiled. This is called a dynamic type system. For example, consider the **squareit** function we defined before:

```
In [44]: def squareit(x):
             return x**2
```

The type of x is not declared in the function definition. We can call it with real or integer inputs, and Python will know what to do:

```
In [45]: squareit(5)

Out[45]: 25

In [46]: squareit(5.5)

Out[46]: 30.25
```

Now suppose the type of the input is a floating-point number. We can write another version of **squareit** that specifies the type.

```
In [47]: def typesquareit(x: float):
             return x**2
```

The input x is now statically typed. However, the purpose here is to add an annotation to remind the users of the input type that should be used. In fact, Python interpreter will not perform any type checking automatically, unless some additional packages are used. In other words, there will be no difference between the *squareit* and *typesquareit* functions.

```
In [48]: typesquareit(5.5)

Out[48]: 30.25

In [49]: typesquareit(5)

Out[49]: 25
```

It can be seen that the function *typesquareit* has no problem taking the integer 5 as an input.

Control flow

Let's create a NumPy array of 10 entries of floating-type. A simple way to do it is by using the function **np.zeros(n)**, which creates an array of size n, and sets each entry to zero. (A similar function is **np.ones(n)** which creates an array of size n with each entry set to 1.)

```
In [50]: values = np.zeros(10)
         values

Out[50]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Now we will set the elements of the array to values of sin function.

```
In [51]: for n in range(10):
             values[n] = np.sin((n+1)**2)

In [52]: values

Out[52]: array([ 0.84147098, -0.7568025 ,  0.41211849, -0.28790332, -0.13235175,
                -0.99177885, -0.95375265,  0.92002604, -0.62988799, -0.50636564])
```

Here is another way to do this. Start with creating an empty array:

```
In [53]: newvalues = np.array([])
```

Then use a **while** statement to generate the values, and append them to the array.

```
In [54]: n = 1
         while n<=10:
             newvalues = np.append(values, np.sin(n**2))
             n += 1
         newvalues

Out[54]: array([ 0.84147098, -0.7568025 ,  0.41211849, -0.28790332, -0.13235175,
                -0.99177885, -0.95375265,  0.92002604, -0.62988799, -0.50636564,
                -0.50636564])
```

Here is how the **if** statement works:

```
In [55]: def f(x, y):
             if x < y:
                 print(x, ' is less than ', y)
             elif x > y:
                 print(x, ' is greater than ', y)
             else:
                 print(x, ' is equal to ', y)

In [56]: f(2, 3)

2  is less than  3


In [57]: f(3, 2)

3  is greater than  2


In [58]: f(1, 1)

1  is equal to  1
```

In the next example we use **if** and **while** to find all the odd numbers in $\{1, \ldots, 10\}$. The empty array created in the first line is of int64 type.

```
In [59]: odds = np.array([]).astype('int64')
         n = 1
         while n <= 10:
             if n%2 == 1:
                 odds = np.append(odds, n)
             n += 1
         odds
```

```
Out[59]: array([1, 3, 5, 7, 9])
```

Here is an interesting property of the function **break**:

```
In [60]: n = 1
         while n <= 20:
             if n%2 == 0:
                 print(n)
                 break
             n += 1
```

```
2
```

Why did the above execution stop at 2? Let's try removing **break**:

```
In [61]: n = 1
         while n <= 20:
             if n%2 == 0:
                 print(n)
             n += 1
```

```
2
4
6
8
10
12
14
16
18
20
```

The function **break** causes the code to exit the while loop once it is evaluated.

Random numbers

These are 5 uniform random numbers from (0,1).

```
In [62]: import numpy as np
```

```
In [63]: np.random.rand(5)
```

```
Out[63]: array([0.43296376, 0.66060029, 0.91030457, 0.27065652, 0.58976353])
```

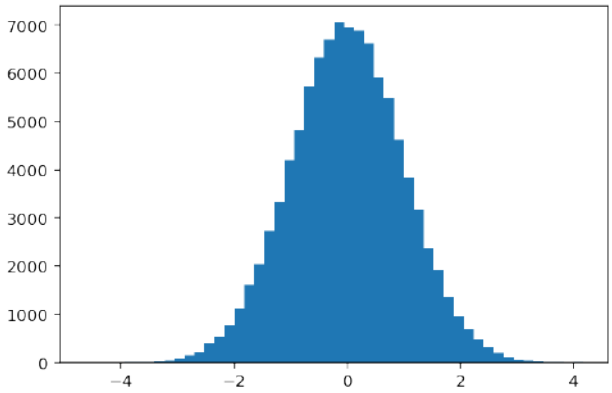
And these are random numbers from the standard normal distribution:

```
In [64]: np.random.randn(5)
```

```
Out[64]: array([-0.60385045, -0.1138149 ,  0.80330871, -0.85704952,  1.26651986])
```

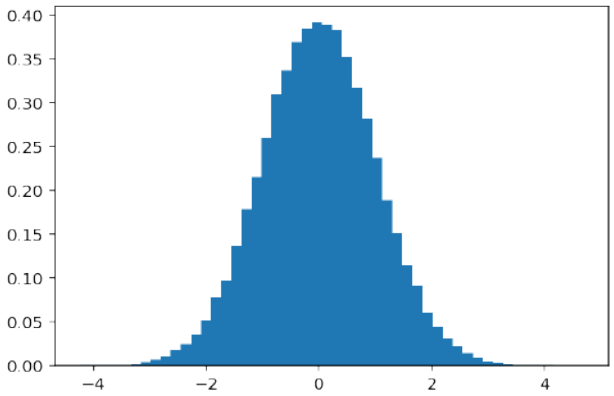
Here is a frequency histogram of 10^5 random numbers from the standard normal distribution using 50 bins:

```
In [65]: y = np.random.randn(10**5)
         plt.hist(y, 50);
```



Sometimes we are interested in relative frequency histograms where the height of each bin is the relative frequency of the numbers in the bin. Adding the option "density=true" outputs a relative frequency histogram:

```
In [66]: y = np.random.randn(10**5)
         plt.hist(y, 50, density=True);
```



Exercise 1.2-1: In Python you can compute the factorial of a positive integer n by the built-in function **factorial(n)** from the subpackage *special* in SciPy (**scipy.special.factorial(n)**). Write your own version of this function, called $factorial2$, using a for loop. Use the **time.time()** function to compare the execution time of your version and the built-in version of the factorial function.

Exercise 1.2-2: Write a Python code to estimate the value of π using the following procedure: Place a circle of diameter one in the unit square. Generate 10,000 pairs of random numbers (u, v) from the unit square. Count the number of pairs (u, v) that fall into the circle, and call this number n. Then $n/10000$ is approximately the area of the circle. (This approach is known as the Monte Carlo method.)

Exercise 1.2-3: Consider the following function

$$f(x,n) = \sum_{i=1}^{n} \prod_{j=1}^{i} x^{n-j+1}.$$

a) Compute $f(2,3)$ by hand.

b) Write a Python code that computes f. Verify $f(2,3)$ matches your answer above.

1.3 Computer arithmetic

The way computers store numbers and perform computations could surprise the beginner. In Python if you type $(\sqrt{3})^2$ the result will be 2.9....96, where 9 is repeated 15 times. Here are two obvious but fundamental differences in the way computers do arithmetic:

- only finitely many numbers can be represented in a computer;
- a number represented in a computer can only have finitely many digits.

Therefore the numbers that can be represented in a computer exactly is only a subset of rational numbers. Anytime the computer performs an operation whose outcome is not a number that can be represented exactly in the computer, an approximation will replace the exact number. This is called the *roundoff error*: error produced when a computer is used to perform real number calculations.

Floating-point representation of real numbers

Here is a general model for representing real numbers in a computer:

$$x = s(.a_1 a_2 ... a_t)_\beta \times \beta^e \tag{1.1}$$

where

$$\begin{aligned} s &\to \text{sign of } x = \pm 1 \\ e &\to \text{exponent, with bounds } L \leq e \leq U \\ (.a_1...a_t)_\beta &= \frac{a_1}{\beta} + \frac{a_2}{\beta^2} + ... + \frac{a_t}{\beta^t};\ \text{the mantissa} \\ \beta &\to \text{base} \\ t &\to \text{number of digits; the precision.} \end{aligned}$$

In the floating-point representation (1.1), if we specify e in such a way that $a_1 \neq 0$, then the representation will be unique. This is called the **normalized** floating-point representation. For example if $\beta = 10$, in the normalized floating-point we would write 0.012 as 0.12×10^{-1}, instead of choices like 0.012×10^0 or 0.0012×10.

In most computers today, the base is $\beta = 2$. Bases 8 and 16 were used in old IBM mainframes in the past. Some handheld calculators use base 10. An interesting historical example is a short-lived computer named Setun developed at Moscow State University which used base 3.

There are several choices to make in the general floating-point model (1.1) for the values of s, β, t, e. The IEEE 64-bit floating-point representation is the specific model used in most computers today:

$$x = (-1)^s (1.a_2a_3...a_{53})_2 2^{e-1023}. \tag{1.2}$$

Some comments:

- Notice how s appears in different forms in equations (1.1) and (1.2). In (1.2), s is either 0 or 1. If $s = 0$, then x is positive. If $s = 1$, x is negative.

- Since $\beta = 2$, in the normalized floating-point representation of x the first (nonzero) digit after the decimal point has to be 1. Then we do not have to store this number. That's why we write x as a decimal number starting at 1 in (1.2). Even though precision is $t = 52$, we are able to access up to the 53rd digit a_{53}.

- The bounds for the exponent are: $0 \leq e \leq 2047$. We will discuss where 2047 comes from shortly. But first, let's discuss why we have $e-1023$ as the exponent in the representation (1.2), as opposed to simply e (which we had in the representation (1.1)). If the smallest exponent possible was $e = 0$, then the smallest positive number the computer can generate would be $(1.00...0)_2 = 1$: certainly we need the computer to represent numbers less than 1! That's why we use the shifted expression $e - 1023$, called the **biased exponent**, in the representation (1.2). Note that the bounds for the biased exponent are $-1023 \leq e - 1023 \leq 1024$.

Here is a schema that illustrates how the physical bits of a computer correspond to the representation

above. Each cell in the table below, numbered 1 through 64, correspond to the physical bits in the computer memory.

1	2	3	...	12	13	...	64

- The first bit is the sign bit: it stores the value for s, 0 or 1.
- The blue bits 2 through 12 store the exponent e (not $e-1023$). Using 11 bits, one can generate the integers from 0 to $2^{11}-1=2047$. Here is how you get the smallest and largest values for e:

$$e=(00...0)_2=0$$
$$e=(11...1)_2=2^0+2^1+...+2^{10}=\frac{2^{11}-1}{2-1}=2047.$$

- The red bits, and there are 52 of them, store the digits a_2 through a_{53}.

Example 9. Find the floating-point representation of 10.375.

Solution. You can check that $10=(1010)_2$ and $0.375=(.011)_2$ by computing

$$10=0\times 2^0+1\times 2^1+0\times 2^2+1\times 2^3$$
$$0.375=0\times 2^{-1}+1\times 2^{-2}+1\times 2^{-3}.$$

Then

$$10.375=(1010.011)_2=(1.010011)_2\times 2^3$$

where $(1.010011)_2\times 2^3$ is the normalized floating-point representation of the number. Now we rewrite this in terms of the representation (1.2):

$$10.375=(-1)^0(1.010011)_2\times 2^{1026-1023}.$$

Since $1026=(10000000010)_2$, the bit by bit representation is:

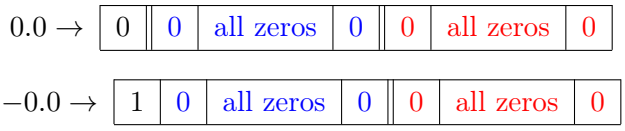
0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	...	0

Notice the first sign bit is 0 since the number is positive. The next 11 bits (in blue) represent the exponent $e=1026$, and the next group of red bits are the mantissa, filled with 0's after the last digit of the mantissa. In Python, although there is no built-in function that produces the bit by bit representation of a number, we can define the following function named *float2bin*, which provides the bit representation of a floating-point number, based on the *struct* package:

```
In [1]: import struct
        def float2bin(f):
            (d,) = struct.unpack(">Q", struct.pack(">d", f))
            return f'{d:064b}'

In [2]: float2bin(10.375)

Out[2]: '0100000000100100110000000000000000000000000000000000000000000000'
```

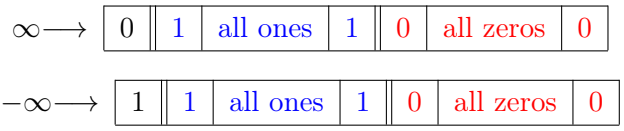
Special cases: zero, infinity, NAN

In the floating-point arithmetic there are two zeros: $+0.0$ and -0.0, and they have special representations. In the representation of zero, all exponent and mantissa bits are set to 0. The sign bit is 0 for $+0.0$, and 1, for -0.0:



When the exponent bits are set to zero, we have $e = 0$ and thus $e - 1023 = -1023$. This arrangement, all exponent bits set to zero, is reserved for ± 0.0 and **subnormal numbers**. Subnormal numbers are an exception to our normalized floating-point representation, an exception that is useful in various ways. For details see Goldberg [9].

Here is how plus and minus infinity is represented in the computer:



When the exponent bits are set to one, we have $e = 2047$ and thus $e - 1023 = 1024$. This arrangement is reserved for $\pm\infty$ as well as other special values such as NaN (not-a-number).

In conclusion, even though $-1023 \leq e-1023 \leq 1024$ in (1.2), when it comes to representing non-zero real numbers, we only have access to exponents in the following range: $-1022 \leq e-1023 \leq 1023$.

Therefore, the smallest positive real number that can be represented in a computer is

$$x = (-1)^0(1.00\ldots0)_2 \times 2^{-1022} = 2^{-1022} \approx 0.2 \times 10^{-307}$$

and the largest is

$$\begin{aligned} x = (-1)^0(1.11\ldots1)_2 \times 2^{1023} &= \left(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{52}}\right) \times 2^{1023} \\ &= (2 - 2^{-52})2^{1023} \\ &\approx 0.18 \times 10^{309}. \end{aligned}$$

During a calculation, if a number less than the smallest floating-point number is obtained, then we obtain the **underflow error**. A number greater than the largest gives **overflow error**.

Exercise 1.3-1: Consider the following toy model for a normalized floating-point representation in base 2: $x = (-1)^s(1.a_2a_3)_2 \times 2^e$ where $-1 \leq e \leq 1$. Find all positive machine numbers (there are 12 of them) that can be represented in this model. Convert the numbers to base 10, and then carefully plot them on the number line, by hand, and comment on how the numbers are spaced.

Representation of integers

In the previous section, we discussed representing real numbers in a computer. Here we will give a brief discussion of representing integers. How does a computer represent an integer n? As in real numbers, we start with writing n in base 2. We have 64 bits to represent its digits and sign. As in the floating-point representation, we can allocate one bit for the sign, and use the rest, 63 bits, for the digits. This approach has some disadvantages when we start adding integers. Another approach, known as the **two's complement**, is more commonly used, including in Python.

For an example, assume we have 8 bits in our computer. To represent 12 in two's complement (or any positive integer), we simply write it in its base 2 expansion: $(00001100)_2$. To represent -12, we do the following: flip all digits, replacing 1 by 0, and 0 by 1, and then add 1 to the result. When we flip digits for 12, we get $(11110011)_2$, and adding 1 (in binary), gives $(11110100)_2$. Therefore -12 is represented as $(11110100)_2$ in two's complement approach. It may seem mysterious to go through all this trouble to represent -12, until you add the representations of 12 and -12,

$$(00001100)_2 + (11110100)_2 = (100000000)_2$$

and realize that the first 8 digits of the sum (from right to left), which is what the computer can only represent (ignoring the red digit 1), is $(00000000)_2$. So just like $12 + (-12) = 0$ in base 10, the sum of the representations of these numbers is also 0.

We can repeat these calculations with 64-bits, using Python. The function *int2bin* defined below outputs the digits of an integer, using two's complement for negative numbers:

```
In [1]: import struct
        def int2bin(i):
            (d,) = struct.unpack(">Q", struct.pack(">q", i))
            return f'{d:064b}'

In [2]: int2bin(12)
```

```
Out[2]: '00000000000000000000000000000000000000000000000000000000000001100'

In [3]: int2bin(-12)

Out[3]: '11111111111111111111111111111111111111111111111111111111110100'
```

You can verify that the sum of these representations is 0, when truncated to 64-digits.

Here is another example illustrating the advantages of two's complement. Consider -3 and 5, with representations,

$$-3 = (11111101)_2 \text{ and } 5 = (00000101)_2.$$

The sum of -3 and 5 is 2; what about the binary sum of their representations? We have

$$(11111101)_2 + (00000101)_2 = (100000010)_2$$

and if we ignore the ninth bit in red, the result is $(10)_2$, which is indeed 2. Notice that if we followed the same approach used in the floating-point representation and allocated the leftmost bit to the sign of the integer, we would not have had this property.

We will not discuss integer representations and integer arithmetic any further. However one useful fact to keep in mind is the following: in two's complement, using 64 bits, one can represent integers between $-2^{63} = -9223372036854775808$ and $2^{63} - 1 = 9223372036854775807$. Any integer below or above yields **underflow** or **overflow** error. However, Python is special compared to the other programming languages in that it supports arbitrary-precision integer implementation. Integers of any size can be supported in Python, while it is not true for floats.

Example 10. From Calculus, we know that $\lim_{n->\infty} \frac{n^n}{n!} = \infty$. Therefore, computing $\frac{n^n}{n!}$, which is a float, for large n will cause overflow at some point. Here is a Python code for this calculation:

```
In [1]: from scipy.special import factorial
        f = lambda n: n**n/factorial(n)
```

Let's have a closer look at this function. The Python function **factorial(n)** computes the factorial, returned as a float, as the gamma function at $n + 1$. If we call f with an integer input, then the above code will compute n^n as an integer and factorial(n) as a float. Then it will divide these numbers, by first converting n^n to a float, and obtain a floating-point factorial. Let's compute $f(n) = \frac{n^n}{n!}$ as $n = 1, ..., 1000$:

```
In [2]: for n in range(1, 1001):
            print(n, f(n))
```

.

.

```
.
135 1.4629522660965042e+57
136 3.962087287769907e+57
137 1.0730739055834461e+58
138 2.90634235201109e+58
139 7.871822311096416e+58
140 2.132136506640898e+59
141 5.775183434192781e+59
142 1.5643266954396485e+60
143 4.2374040203696554e+60
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)

<ipython-input-2-88a424afb432> in <module>
  1 for n in range(1, 1001):
----> 2     print(n, f(n))


<ipython-input-1-6df979c5f68c> in <lambda>(n)
  1 from scipy.special import factorial
----> 2 f = lambda n: n**n/factorial(n)


OverflowError: int too large to convert to float
```

Notice that the process cannot proceed beyond $n = 143$. Where exactly does the error occur? Python can compute 144^{144} exactly (arbitrary-precision); however, when it is converted to a floating-point number, overflow occurs. For 143^{143}, overflow does not happen.

```
In [3]: float(143**143)

Out[3]: 1.6332525972973913e+308

In [4]: float(144**144)
```

```
---------------------------------------------------------------------------

OverflowError                                Traceback (most recent call last)

<ipython-input-2-9d2f48a83404> in <module>
----> 1 float(144**144)


OverflowError: int too large to convert to float
```

The function $f(n)$ can be coded in a much better way if we rewrite $\frac{n^n}{n!}$ as $\frac{n}{n}\frac{n}{n-1}...\frac{n}{1}$. Each fraction can be computed separately, and then multiplied, which will slow down the growth of the numbers. Here is a new code using a **for** statement.

```
In [5]: def f(n):
            pr = 1.
            for i in range(1, n):
                pr *= n/(n-i)
            return pr
```

Let's compute $f(n) = \frac{n^n}{n!}$ as $n = 1, ..., 1000$ again:

```
In [6]: for i in range(1, 1001):
            print(i, f(i))
```

```
.
.
.
705 2.261381911989747e+304
706 6.142719392952918e+304
707 1.6685832310803885e+305
708 4.532475935558631e+305
709 1.2311857272492938e+306
710 3.34435267514059e+306
711 9.084499321839277e+306
712 2.4676885942863887e+307
713 6.70316853566547e+307
714 inf
```

```
715 inf
716 inf
717 inf
718 inf
.
.
.
```

The previous version of the code gave overflow error when $n = 144$. This version has no difficulty in computing $n^n/n!$ for $n = 144$. In fact, we can go as high as $n = 713$. Overflow in floating-point arithmetic yields the output inf, which stands for infinity.

Another way to accommodate a larger value of n is to define the function f in this way:

```
In [7]: from scipy.special import factorial
        f = lambda n: n**n/factorial(n, True)
```

With the addition input "True", the function **factorial(n)** returns an integer with arbitrary precision, instead of a float, so that both the numerator and denominator are integers. The limit process stops only when the division produces a floating-point number that is so large that overflow occurs. As a result, the same result as in the improved algorithm above is expected. Please verify that the process can continue until $n = 713$ with the modified f function.

We will discuss several features of computer arithmetic in the rest of this section. The discussion is easier to follow if we use the familiar base 10 representation of numbers instead of base 2. To this end, we introduce the **normalized decimal floating-point representation**:

$$\pm 0.d_1 d_2 ... d_k \times 10^n$$

where $1 \leq d_1 \leq 9, 0 \leq d_i \leq 9$ for all $i = 2, 3, ..., k$. Informally, we call these numbers $k-$digit decimal machine numbers.

Chopping & Rounding

Let x be a real number with more digits the computer can handle: $x = 0.d_1 d_2 \ldots d_k d_{k+1} \ldots \times 10^n$. How will the computer represent x? Let's use the notation $fl(x)$ for the floating-point representation of x. There are two choices, chopping and rounding:

- In chopping, we simply take the first k digits and ignore the rest: $fl(x) = 0.d_1 d_2 \ldots d_k$.
- In rounding, if $d_{k+1} \geq 5$ we add 1 to d_k to obtain $fl(x)$. If $d_{k+1} < 5$, then we simply do as in chopping.

Example 11. Find 5-digit ($k = 5$) chopping and rounding values of the numbers below:

- $\pi = 0.314159265... \times 10^1$
  Chopping gives $fl(\pi) = 0.31415$ and rounding gives $fl(\pi) = 0.31416$.
- 0.0001234567
  We need to write the number in the normalized representation first as 0.1234567×10^{-3}. Now chopping gives 0.12345 and rounding gives 0.12346.

Absolute and relative error

Since computers only give approximations to real numbers, we need to be clear on how we measure the error of an approximation.

Definition 12. Suppose x^* is an approximation to x.

- $|x^* - x|$ is called the **absolute error**
- $\frac{|x^*-x|}{|x|}$ is called the **relative error** ($x \neq 0$)

Relative error usually is a better choice of measure, and we need to understand why.

Example 13. Find absolute and relative errors of

1. $x = 0.20 \times 10^1, x^* = 0.21 \times 10^1$
2. $x = 0.20 \times 10^{-2}, x^* = 0.21 \times 10^{-2}$
3. $x = 0.20 \times 10^5, x^* = 0.21 \times 10^5$

Notice how the only difference in the three cases is the exponent of the numbers. The absolute errors are: 0.01×10 , 0.01×10^{-2} , 0.01×10^5. The absolute errors are different since the exponents are different. However, the relative error in each case is the same: 0.05.

Definition 14. The number x^* is said to approximate x to s significant digits (or figures) if s is the largest nonnegative integer such that

$$\frac{|x - x^*|}{|x|} \leq 5 \times 10^{-s}.$$

In Example 13 we had $\frac{|x-x^*|}{|x|} = 0.05 \leq 5 \times 10^{-2}$ but not less than or equal to 5×10^{-3}. Therefore we say $x^* = 0.21$ approximates $x = 0.20$ to 2 significant digits (but not to 3 digits).

When the computer approximates a real number x by $fl(x)$, what can we say about the error? The following result gives an upper bound for the relative error.

Lemma 15. *The relative error of approximating x by $fl(x)$ in the k-digit normalized decimal floating-point representation satisfies*

$$\frac{|x - fl(x)|}{|x|} \leq \begin{cases} 10^{-k+1} & \text{if chopping} \\ \frac{1}{2}(10^{-k+1}) & \text{if rounding.} \end{cases}$$

Proof. We will give the proof for chopping; the proof for rounding is similar but tedious. Let

$$x = 0.d_1d_2...d_kd_{k+1}... \times 10^n.$$

Then

$$fl(x) = 0.d_1d_2...d_k \times 10^n$$

if chopping is used. Observe that

$$\frac{|x - fl(x)|}{|x|} = \frac{0.d_{k+1}d_{k+2}... \times 10^{n-k}}{0.d_1d_2... \times 10^n} = \left(\frac{0.d_{k+1}d_{k+2}...}{0.d_1d_2...}\right) 10^{-k}.$$

We have two simple bounds: $0.d_{k+1}d_{k+2}... < 1$ and $0.d_1d_2... \geq 0.1$, the latter true since the smallest d_1 can be, is 1. Using these bounds in the equation above we get

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{0.1}10^{-k} = 10^{-k+1}.$$

□

Remark 16. Lemma 15 easily transfers to the base 2 floating-point representation: $x = (-1)^s(1.a_2...a_{53})_2 \times 2^{e-1023}$, by

$$\frac{|x - fl(x)|}{|x|} \leq \begin{cases} 2^{-t+1} = 2^{-53+1} = 2^{-52} & \text{if chopping} \\ \frac{1}{2}(2^{-t+1}) = 2^{-53} & \text{if rounding.} \end{cases}$$

Machine epsilon

Machine epsilon ϵ is the smallest positive floating point number for which $fl(1+\epsilon) > 1$. This means, if we add to 1.0 any number less than ϵ, the machine computes the sum as 1.0.

The number 1.0 in its binary floating-point representation is simply $(1.0\ldots0)_2$ where $a_2 = a_3 = ... = a_{53} = 0$. We want to find the smallest number that gives a sum larger than 1.0, when it is added to 1.0. The answer depends on whether we chop or round.

If we are chopping, examine the binary addition

$$\begin{array}{cccccc} & & a_2 & & a_{52} & a_{53} \\ & 1. & 0 & \ldots & 0 & 0 \\ + & 0. & 0 & \ldots & 0 & 1 \\ \hline & 1. & 0 & \ldots & 0 & 1 \end{array}$$

and notice $(0.0...01)_2 = \left(\frac{1}{2}\right)^{52} = 2^{-52}$ is the smallest number we can add to 1.0 such that the sum will be different than 1.0.

If we are rounding, examine the binary addition

$$\begin{array}{cccccc} & a_2 & & a_{52} & a_{53} & \\ 1. & 0 & \ldots & 0 & 0 & \\ + \; 0. & 0 & \ldots & 0 & 0 & 1 \\ \hline 1. & 0 & \ldots & 0 & 0 & 1 \end{array}$$

where the sum has to be rounded to 53 digits to obtain

$$\begin{array}{ccccc} a_2 & & a_{52} & a_{53} \\ 1. \; 0 & \ldots & 0 & 1 \end{array}.$$

Observe that the number added to 1.0 above is $(0.0...01)_2 = \left(\frac{1}{2}\right)^{53} = 2^{-53}$, which is the smallest number that will make the sum larger than 1.0 with rounding.

In summary, we have shown

$$\epsilon = \begin{cases} 2^{-52} & \text{if chopping} \\ 2^{-53} & \text{if rounding} \end{cases}.$$

As a consequence, notice that we can restate the inequality in Remark 16 in a compact way using the machine epsilon as:

$$\frac{|x - fl(x)|}{|x|} \leq \epsilon.$$

Remark 17. There is another definition of machine epsilon: it is the distance between 1.0 and the next floating-point number.

$$\begin{array}{cccccc} & & a_2 & & a_{52} & a_{53} \\ \text{number 1.0} & 1. & 0 & \ldots & 0 & 0 \\ \text{next number} & 1. & 0 & \ldots & 0 & 1 \\ \hline \text{distance} & 0. & 0 & \ldots & 0 & 1 \end{array}$$

Note that the distance (absolute value of the difference) is $\left(\frac{1}{2}\right)^{52} = 2^{-52}$. In this alternative definition, machine epsilon is not based on whether rounding or chopping is used. Also, note that the distance between two adjacent floating-point numbers is not constant, but it is smaller for smaller values, and larger for larger values (see Exercise 1.3-1).

Propagation of error

We discussed the resulting error when chopping or rounding is used to approximate a real number by its machine version. Now imagine carrying out a long calculation with many arithmetical operations, and at each step there is some error due to say, rounding. Would all the rounding errors accumulate and cause havoc? This is a rather difficult question to answer in general. For a much simpler example, consider adding two real numbers x, y. In the computer, the numbers are represented as $fl(x), fl(y)$. The sum of these number is $fl(x) + fl(y)$; however, the computer can only represent its floating-point version, $fl(fl(x) + fl(y))$. Therefore the relative error in adding two numbers is:

$$\left| \frac{(x+y) - fl(fl(x) + fl(y))}{x+y} \right|.$$

In this section, we will look at some specific examples where roundoff error can cause problems, and how we can avoid them.

Subtraction of nearly equal quantities: Cancellation of leading digits

The best way to explain this phenomenon is by an example. Let $x = 1.123456, y = 1.123447$. We will compute $x - y$ and the resulting roundoff error using rounding and 6-digit arithmetic. First, we find $fl(x), fl(y)$:

$$fl(x) = 1.12346, fl(y) = 1.12345.$$

The absolute and relative error due to rounding is:

$$|x - fl(x)| = 4 \times 10^{-6}, |y - fl(y)| = 3 \times 10^{-6}$$
$$\frac{|x - fl(x)|}{|x|} = 3.56 \times 10^{-6}, \frac{|y - fl(y)|}{|y|} = 2.67 \times 10^{-6}.$$

From the relative errors, we see that $fl(x)$ and $fl(y)$ approximate x and y to six significant digits. Let's see how the error propagates when we subtract x and y. The actual difference is:

$$x - y = 1.123456 - 1.123447 = 0.000009 = 9 \times 10^{-6}.$$

The computer finds this difference first by computing $fl(x), fl(y)$, then taking their difference and approximating the difference by its floating-point representation: $fl(fl(x) - fl(y))$:

$$fl(fl(x) - fl(y)) = fl\,(1.12346 - 1.12345) = 10^{-5}.$$

The resulting absolute and relative errors are:

$$|(x-y)-(fl(fl(x)-fl(y)))| = 10^{-6}$$
$$\frac{|(x-y)-(fl(fl(x)-fl(y)))|}{|x-y|} = 0.1.$$

Notice how large the relative error is compared to the absolute error! The machine version of $x-y$ approximates $x-y$ to only one significant digit. Why did this happen? When we subtract two numbers that are nearly equal, the leading digits of the numbers cancel, leaving a result close to the rounding error. In other words, the rounding error dominates the difference.

Division by a small number

Let $x = 0.444446$ and compute $\frac{x}{10^{-5}}$ in a computer with 5-digit arithmetic and rounding. We have $fl(x) = 0.44445$, with an absolute error of 4×10^{-6} and relative error of 9×10^{-6}. The exact division is $\frac{x}{10^{-5}} = 0.444446\times10^{5}$. The computer computes: $fl\left(\frac{x}{10^{-5}}\right) = 0.44445\times10^{5}$, which has an absolute error of 0.4 and relative error of 9×10^{-6}. The absolute error went from 4×10^{-6} to 0.4. Perhaps not surprisingly, division by a small number magnifies the absolute error but not the relative error.

Consider the computation of

$$\frac{1-\cos x}{\sin x}$$

when x is near zero. This is a problem where we have both *subtraction of nearly equal quantities* which happens in the numerator, and *division by a small number*, when x is close to zero. Let $x = 0.1$. Continuing with five-digit rounding, we have

$$fl(\sin 0.1) = 0.099833,\ fl(\cos 0.1) = 0.99500$$
$$fl\left(\frac{1-\cos 0.1}{\sin 0.1}\right) = 0.050084.$$

The exact result to 8 digits is 0.050041708, and the relative error of this computation is 8.5×10^{-4}. Next we will see how to reduce this error using a simple algebraic identity.

Ways to avoid loss of accuracy

Here we will discuss some examples where a careful rewriting of the expression to compute can make the roundoff error much smaller.

Example 18. Let's revisit the calculation of

$$\frac{1-\cos x}{\sin x}.$$

Observe that using the algebraic identity

$$\frac{1-\cos x}{\sin x} = \frac{\sin x}{1+\cos x}$$

removes both difficulties encountered before: there is no cancellation of significant digits or division by a small number. Using five-digit rounding, we have

$$fl\left(\frac{\sin 0.1}{1+\cos 0.1}\right) = 0.050042.$$

The relative error is 5.8×10^{-6}, about a factor of 100 smaller than the error in the original computation.

Example 19. Consider the quadratic formula: the solution of $ax^2 + bx + c = 0$ is

$$r_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}, r_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}.$$

If $b \approx \sqrt{b^2-4ac}$, then we have a potential loss of precision in computing one of the roots due to cancellation. Let's consider a specific equation: $x^2 - 11x + 1 = 0$. The roots from the quadratic formula are: $r_1 = \frac{11+\sqrt{117}}{2} \approx 10.90832691$, and $r_2 = \frac{11-\sqrt{117}}{2} \approx 0.09167308680$.

Next will use four-digit arithmetic with rounding to compute the roots:

$$fl(\sqrt{117}) = 10.82$$
$$fl(r_1) = fl\left(\frac{fl(fl(11.0)+fl(\sqrt{117}))}{fl(2.0)}\right) = fl\left(\frac{fl(11.0+10.82)}{2.0}\right) = fl\left(\frac{21.82}{2.0}\right) = 10.91$$
$$fl(r_2) = fl\left(\frac{fl(fl(11.0)-fl(\sqrt{117}))}{fl(2.0)}\right) = fl\left(\frac{fl(11.0-10.82)}{2.0}\right) = fl\left(\frac{0.18}{2.0}\right) = 0.09.$$

The relative errors are:

$$\text{rel error in } r_1 = \left|\frac{10.90832691-10.91}{10.90832691}\right| = 1.5 \times 10^{-4}$$
$$\text{rel error in } r_2 = \left|\frac{0.09167308680-0.09}{0.09167308680}\right| = 1.8 \times 10^{-2}.$$

Notice the larger relative error in r_2 compared to that of r_1, about a factor of 100, which is due to cancellation of leading digits when we compute $11.0 - 10.82$.

One way to fix this problem is to rewrite the offending expression by rationalizing the numerator:

$$r_2 = \frac{11.0-\sqrt{117}}{2} = \left(\frac{1}{2}\right)\frac{11.0-\sqrt{117}}{11.0+\sqrt{117}}\left(11.0+\sqrt{117}\right) = \left(\frac{1}{2}\right)\frac{4}{11.0+\sqrt{117}} = \frac{2}{11.0+\sqrt{117}}.$$

If we use this formula to compute r_2 we get:

$$fl(r_2) = fl\left(\frac{2.0}{fl(11.0 + fl(\sqrt{117}))}\right) = fl\left(\frac{2.0}{21.82}\right) = 0.09166.$$

The new relative error in r_2 is:

$$\text{rel error in } r_2 \ = \left(\frac{0.09167308680 - 0.09166}{0.09167308680}\right) = 1.4 \times 10^{-4},$$

an improvement about a factor of 100, even though in the new way of computing r_2 there are two operations where rounding error happens instead of one.

Example 20. The simple procedure of adding numbers, even if they do not have mixed signs, can accumulate large errors due to rounding or chopping. Several sophisticated algorithms to add large lists of numbers with accumulated error smaller than straightforward addition exist in the literature (see, for example, Higham [11]).

For a simple example, consider using four-digit arithmetic with rounding, and computing the average of two numbers, $\frac{a+b}{2}$. For $a = 2.954$ and $b = 100.9$, the true average is 51.927. However, four-digit arithmetic with rounding yields:

$$fl\left(\frac{100.9 + 2.954}{2}\right) = fl\left(\frac{fl(103.854)}{2}\right) = fl\left(\frac{103.9}{2}\right) = 51.95,$$

which has a relative error of 4.43×10^{-4}. If we rewrite the averaging formula as $a + \frac{b-a}{2}$, on the other hand, we obtain 51.93, which has a much smaller relative error of 5.78×10^{-5}. The following table displays the exact and 4-digit computations, with the corresponding relative error at each step.

	a	b	$a+b$	$\frac{a+b}{2}$	$b-a$	$\frac{b-a}{2}$	$a+\frac{b-a}{2}$
4-digit rounding	2.954	100.9	103.9	51.95	97.95	48.98	51.93
Exact			103.854	51.927	97.946	48.973	51.927
Relative error			4.43e-4	4.43e-4	4.08e-5	1.43e-4	5.78e-5

Example 21. There are two standard formulas given in textbooks to compute the sample variance s^2 of the numbers $x_1, ..., x_n$:

1. $s^2 = \frac{1}{n-1}\left[\sum_{i=1}^n x_i^2 - \frac{1}{n}\left(\sum_{i=1}^n x_i\right)^2\right]$,
2. First compute $\bar{x} = \frac{1}{n}\sum_{i=1}^n x_i$, and then $s^2 = \frac{1}{n-1}\sum_{i=1}^n (x_i - \bar{x})^2$.

Both formulas can suffer from roundoff errors due to adding large lists of numbers if n is large, as mentioned in the previous example. However, the first formula is also prone to error due to cancellation of leading digits (see Chan et al [6] for details).

For an example, consider four-digit rounding arithmetic, and let the data be $1.253, 2.411, 3.174$. The sample variance from formula 1 and formula 2 are, 0.93 and 0.9355, respectively. The exact

value, up to 6 digits, is 0.935562. Formula 2 is a numerically more stable choice for computing the variance than the first one.

Example 22. We have a sum to compute:

$$e^{-7} = 1 + \frac{-7}{1} + \frac{(-7)^2}{2!} + \frac{(-7)^3}{3!} + ... + \frac{(-7)^n}{n!}.$$

The alternating signs make this a potentially error prone calculation.

Python reports the "exact" value for e^{-7} as 0.0009118819655545162. If we use Python to compute this sum with $n = 20$, the result is 0.009183673977218275. Here is the Python code for the calculation:

```
In [1]: import numpy as np
        from scipy.special import factorial

In [2]: sum = 1.0
        for n in range(1, 21):
            sum += (-7)**n/factorial(n)
        sum

Out[2]: 0.009183673977218275
```

This result has a relative error of 9.1. We can avoid this huge error if we simply rewrite the above sum as

$$e^{-7} = \frac{1}{e^7} = \frac{1}{1 + 7 + \frac{7^2}{2!} + \frac{7^3}{3!} + ...}.$$

The Python code for this computation using $n = 20$ is below:

```
In [3]: sum = 1.0
        for n in range(1, 21):
            sum += 7**n/factorial(n)
        sum = 1/sum
        sum

Out[3]: 0.0009118951837867185
```

The result is 0.0009118951837867185, which has a relative error of 1.4×10^{-5}.

Exercise 1.3-2: The x-intercept of the line passing through the points (x_1, y_1) and (x_2, y_2) can be computed using either one of the following formulas:

$$x = \frac{x_1 y_2 - x_2 y_1}{y_2 - y_1}$$

or,

$$x = x_1 - \frac{(x_2 - x_1) y_1}{y_2 - y_1}$$

with the assumption $y_1 \neq y_2$.

a) Show that the formulas are equivalent to each other.

b) Compute the x-intercept using each formula when $(x_1, y_1) = (1.02, 3.32)$ and $(x_2, y_2) = (1.31, 4.31)$. Use three-digit rounding arithmetic.

c) Use Python (or a calculator) to compute the x-intercept using the full-precision of the device (you can use either one of the formulas). Using this result, compute the relative and absolute errors of the answers you gave in part (b). Discuss which formula is better and why.

Exercise 1.3-3: Write two functions in Python to compute the binomial coefficient $\binom{m}{k}$ using the following formulas:

a) $\binom{m}{k} = \frac{m!}{k!(m-k)!}$ ($m!$ is **scipy.special.factorial(m)** in Python.)

b) $\binom{m}{k} = (\frac{m}{k})(\frac{m-1}{k-1}) \times ... \times (\frac{m-k+1}{1})$
Then, experiment with various values for m, k to see which formula causes overflow first.

Exercise 1.3-4: Polynomials can be evaluated in a nested form (also called Horner's method) that has two advantages: the nested form has significantly less computation, and it can reduce roundoff error. For

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1} + a_n x^n$$

its nested form is

$$p(x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-1} + x(a_n))...)).$$

Consider the polynomial $p(x) = x^2 + 1.1x - 2.8$.

a) Compute $p(3.5)$ using three-digit rounding, and three-digit chopping arithmetic. What are the absolute errors? (Note that the exact value of $p(3.5)$ is 13.3.)

b) Write $x^2 + 1.1x - 2.8$ in nested form by these simple steps:

$$x^2 + 1.1x - 2.8 = (x^2 + 1.1x) - 2.8 = (x + 1.1)x - 2.8.$$

Then compute $p(3.5)$ using three-digit rounding and chopping using the nested form. What are the absolute errors? Compare the errors with the ones you found in (a).

Exercise 1.3-5: Consider the polynomial written in standard form: $5x^4+3x^3+4x^2+7x-5$.

a) Write the polynomial in its nested form. (See the previous problem.)

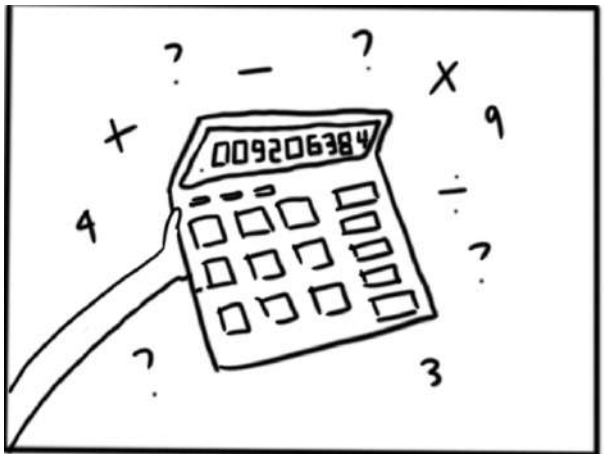
b) How many multiplications does the nested form require when we evaluate the polynomial at a real number? How many multiplications does the standard form require? Can you generalize your answer to any nth degree polynomial?

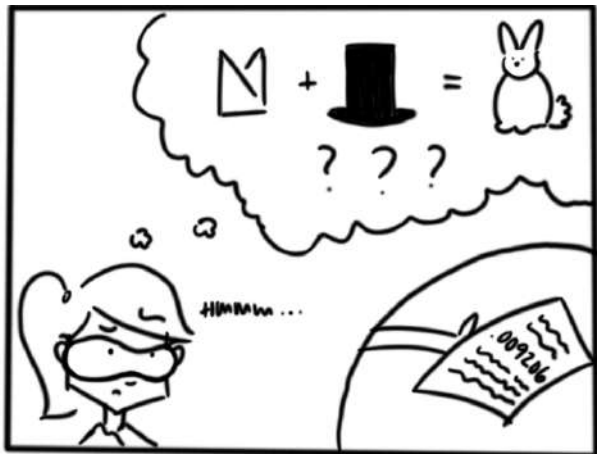
Arya and the unexpected challenges of data analysis

Meet Arya! Arya is a college student interested in math, biology, literature, and acting. Like a typical college student, she texts while walking on campus, complains about demanding professors, and argues in her blogs that homework should be outlawed.

Arya is taking a chemistry class, and she performs some experiments in the lab to find the weight of two substances. Due to difficulty in making precise measurements, she can only assess the weights to four-significant digits of accuracy: 2.312 grams and 0.003982 grams. Arya's professor wants to know the product of these weights, which will be used in a formula.

Arya computes the product using her calculator: $2.312 \times 0.003982 = 0.009206384$, and stares at the result in bewilderment. The numbers she multiplied had four-significant digits, but the product has seven digits! Could this be the result of some magic, like a rabbit hopping out of a magician's hat that was only a handkerchief a moment ago? After some internal deliberations, Arya decides to report the answer to her professor as 0.009206. Do you think Arya was correct in not reporting all of the digits of the product?





Sources of error in applied mathematics

Here is a list of potential sources of error when we solve a problem.

1. Error due to the simplifying assumptions made in the development of a mathematical model for the physical problem.
2. Programming errors.
3. Uncertainty in physical data: error in collecting and measuring data.
4. Machine errors: rounding/chopping, underflow, overflow, etc.
5. Mathematical truncation error: error that results from the use of numerical methods in solving a problem, such as evaluating a series by a finite sum, a definite integral by a numerical integration method, solving a differential equation by a numerical method.

Example 23. The volume of the Earth could be computed using the formula for the volume of a sphere, $V = 4/3\pi r^3$, where r is the radius. This computation involves the following approximations:

1. The Earth is modeled as a sphere (modeling error)
2. Radius $r \approx 6370$ km is based on empirical measurements (uncertainty in physical data)
3. All the numerical computations are done in a computer (machine error)
4. The value of π has to be truncated (mathematical truncation error)

Exercise 1.3-6: The following is from "Numerical mathematics and computing" by Cheney & Kincaid [7]:

In 1996, the Ariane 5 rocket launched by the European Space Agency exploded 40 seconds after lift-off from Kourou, French Guiana. An investigation determined that the horizontal velocity required the conversion of a 64-bit floating-point number to a 16-bit signed integer. It failed because the number was larger than 32,767, which was the largest integer of this type that could be stored in memory. The rocket and its cargo were valued at $500 million.

Search online, or in the library, to find another example of computer arithmetic going very wrong! Write a short paragraph explaining the problem, and give a reference.

Chapter 2

Solutions of equations: Root-finding

Arya and the mystery of the Rhind papyrus

College life is full of adventures, some hopefully of intellectual nature, and Arya is doing her part by taking a history of science class. She learns about the Rhind papyrus; an ancient Egyptian papyrus purchased by an antiquarian named Henry Rhind in Luxor, Egypt, in 1858.

Figure 2.1: Rhind Mathematical Papyrus. (British Museum Image under a Creative Commons license.)

The papyrus has a collection of mathematical problems and their solutions; a translation is given by Chace and Manning [2]. The following is Problem 26, taken from [2]:

A quantity and its 1/4 *added together become 15. What is the quantity?*

Assume 4.

\1	4
\1/4	1
Total	5

As many times as 5 must be multiplied to give 15, so many times 4 must be multiplied to give the required number. Multiply 5 so as to get 15.

\1	5
\2	10
Total	3

Multiply 3 by 4.

1	3
2	6
\4	12

The quantity is

	12
1/4	3
Total	15

Arya's instructor knows she has taken math classes and asks her if she could decipher this solution. Although Arya's initial reaction to this assignment can be best described using the word "despair", she quickly realizes it is not as bad as she thought. Here is her thought process: the question, in our modern notation is, find x if $x + x/4 = 15$. The solution starts with an initial guess $p = 4$. It then evaluates $x + x/4$ when $x = p$, and finds the result to be 5: however, what we need is 15, not 5, and if we multiply both sides of $p + p/4 = 5$ by 3, we get $(3p) + (3p)/4 = 15$. Therefore, the solution is $3p = 12$.

Here is a more general analysis of this solution technique. Suppose we want to solve the equation $g(x) = a$, and that g is a linear map, that is, $g(\lambda x) = \lambda g(x)$ for any constant λ. Then, the solution

is $x = ap/b$ where p is an initial guess with $g(p) = b$. To see this, simply observe

$$g(\frac{ap}{b}) = \frac{a}{b}g(p) = a.$$

The general problem

How can we solve equations that are far complicated than the ancient Egyptians solved? For example, how can we solve $x^2 + 5\cos x = 0$? Stated differently, how can we find the root p such that $f(p) = 0$, where $f(x) = x^2 + 5\cos x$? In this chapter we will learn some iterative methods to solve equations. An **iterative method** produces a sequence of numbers $p_1, p_2, ...$ such that $\lim_{n\to\infty} p_n = p$, and p is the root we seek. Of course, we cannot compute the exact limit, so we stop the iteration at some large N, and use p_N as an approximation to p.

The stopping criteria

With any iterative method, a key question is how to decide when to stop the iteration. How well does p_N approximate p?

Let $\epsilon > 0$ be a small tolerance picked ahead of time. Here are some stopping criteria: stop when

1. $|p_N - p_{N-1}| < \epsilon$,
2. $\left|\frac{p_N - p_{N-1}}{p_N}\right| < \epsilon, p_N \neq 0$, or
3. $|f(p_N)| < \epsilon$.

However, difficulties can arise with any of these criteria:

1. It is possible to have a sequence $\{p_n\}$ such that $p_n - p_{n-1} \to 0$ but $\{p_n\}$ diverges.
2. It is possible to have $|f(p_N)|$ small (called residual) but p_N not close to p.

In our numerical results, we will experiment with various stopping criteria. However, the second criterion is usually preferred over the others.

Exercise 2.-1: Solve the following problems and discuss their relevance to the stopping criteria.

a) Consider the sequence p_n where $p_n = \sum_{i=1}^{n} \frac{1}{i}$. Argue that p_n diverges, but $\lim_{n\to\infty}(p_n - p_{n-1}) = 0$.

b) Let $f(x) = x^{10}$. Clearly, $p = 0$ is a root of f, and the sequence $p_n = \frac{1}{n}$ converges to p. Show that $f(p_n) < 10^{-3}$ if $n > 1$, but to obtain $|p - p_n| < 10^{-3}$, n must be greater than 10^3.

2.1 Error analysis for iterative methods

Assume we have an iterative method $\{p_n\}$ that converges to the root p of some function. How can we assess the rate of convergence?

Definition 24. Suppose $\{p_n\}$ converges to p. If there are constants $C > 0$ and $\alpha > 1$ such that

$$|p_{n+1} - p| \leq C|p_n - p|^{\alpha}, \tag{2.1}$$

for $n \geq 1$, then we say $\{p_n\}$ converges to p with order α.

Special cases:

- If $\alpha = 1$ and $C < 1$, we say the convergence is linear, and the rate of convergence is C. In this case, using induction, we can show

$$|p_{n+1} - p| \leq C^n|p_1 - p|. \tag{2.2}$$

  There are some methods for which Equation (2.2) holds, but Equation (2.1) does not hold for any $C < 1$. We still call these methods to be of linear convergence. An example is the bisection method.

- If $\alpha > 1$, we say the convergence is superlinear. In particular, the case $\alpha = 2$ is called quadratic convergence.

Example 25. Consider the sequences defined by

$$p_{n+1} = 0.7p_n \text{ and } p_1 = 1$$
$$p_{n+1} = 0.7p_n^2 \text{ and } p_1 = 1.$$

The first sequence converges to 0 linearly, and the second quadratically. Here are a few iterations of the sequences:

n	Linear	Quadratic
1	0.7	0.7
4	0.24	4.75×10^{-3}
8	5.76×10^{-2}	3.16×10^{-40}

Observe how fast quadratic convergence is compared to linear convergence.

2.2 Bisection method

Let's recall the Intermediate Value Theorem (IVT), Theorem 6: If a continuous function f defined on $[a, b]$ satisfies $f(a)f(b) < 0$, then there exists $p \in [a, b]$ such that $f(p) = 0$.

Here is the idea behind the method. At each iteration, divide the interval $[a, b]$ into two subintervals and evaluate f at the midpoint. Discard the subinterval that does not contain the root and continue with the other interval.

Example 26. Compute the first three iterations by hand for the function plotted in Figure (2.2).

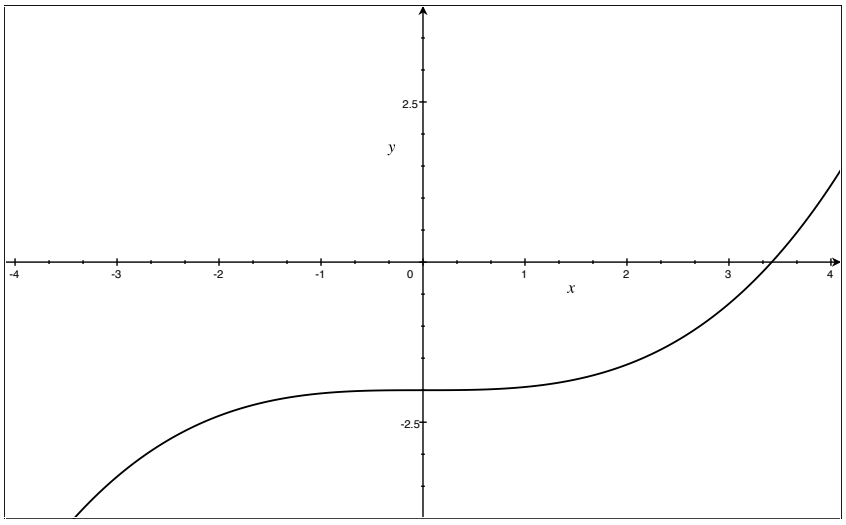


Figure 2.2

Step 1 : To start, we need to pick an interval $[a, b]$ that contains the root, that is, $f(a)f(b) < 0$. From the plot, it is clear that $[0, 4]$ is a possible choice. In the next few steps, we will be working with a sequence of intervals. For convenience, let's label them as $[a, b] = [a_1, b_1], [a_2, b_2], [a_3, b_3]$, etc. Our first interval is then $[a_1, b_1] = [0, 4]$. Next we find the midpoint of the interval, $p_1 = 4/2 = 2$, and use it to obtain two subintervals $[0, 2]$ and $[2, 4]$. Only one of them contains the root, and that is $[2, 4]$.

Step 2: From the previous step, our current interval is $[a_2, b_2] = [2, 4]$. We find the midpoint[1] $p_2 = \frac{2+4}{2} = 3$, and form the subintervals $[2, 3], [3, 4]$. The one that contains the root is $[3, 4]$.

Step 3: We have $[a_3, b_3] = [3, 4]$. The midpoint is $p_3 = 3.5$. We are now pretty close to the root visually, and we stop the calculations!

In this simple example, we did not consider

- Stopping criteria
- It's possible that the stopping criterion is not satisfied in a reasonable amount of time. We need a maximum number of iterations we are willing to run the code with.

Remark 27.

1. A numerically more stable formula to compute the midpoint is $a + \frac{b-a}{2}$ (see Example 20).
2. There is a convenient stopping criterion for the bisection method that was not mentioned before. One can stop when the interval $[a, b]$ at step n is such that $|a - b| < \epsilon$. This is similar to the first stopping criterion discussed earlier, but not the same. One can also use more than one stopping criterion; an example is in the Python code that follows.

Python code for the bisection method

In Example 26, we kept track of the intervals and midpoints obtained from the bisection method, by labeling them as $[a_1, b_1], [a_2, b_2], ...$, and $p_1, p_2, ...$ So at step n of the method, we know we are working on the interval $[a_n, b_n]$ and its midpoint is p_n. This approach will be useful when we study the convergence of the method in the next theorem. However, keeping track of the intervals and midpoints is not needed in the computer code. Instead, in the Python code below, we will let $[a, b]$ be the current interval we are working on, and when we obtain a new interval in the following step, we will simply call the new interval $[a, b]$, overwriting the old one. Similarly, we will call the midpoint p, and update it at each step.

```
In [1]: import numpy as np

In [2]: def bisection(f, a, b, eps, N):
            n = 1
            p = 0. # to ensure the value of p carries out of the while loop
            while n <= N:
                p = a + (b-a)/2
                if np.isclose(f(p), 0) or np.abs(a-b)<eps:
```

[1] Notice how we label the midpoints, as well as the endpoints of the interval, with the step number.

```
            print('p is ', p, ' and the iteration number is ', n)
            return
        if f(a)*f(p) < 0:
            b = p
        else:
            a = p
        n += 1
    y = f(p)
    print('Method did not converge. The last iteration gives ',
          p, ' with function value ', y)
```

Let's use the bisection method to find the root of $f(x) = x^5 + 2x^3 - 5x - 2$, with $\epsilon = 10^{-4}$. Note that $[0, 2]$ contains a root, since $f(0) < 0$ and $f(2) > 0$. We set $N = 20$ below.

```
In [3]: bisection(lambda x: x**5+2*x**3-5*x-2, 0, 2, 1e-4, 20)

p is  1.319671630859375  and the iteration number is  16
```

The value of the function at the estimated root is:

```
In [4]: x = 1.319671630859375
        x**5+2*x**3-5*x-2

Out[4]: 0.000627945623044468
```

Let's see what happens if N is set too small and the method does not converge.

```
In [5]: bisection(lambda x: x**5+2*x**3-5*x-2, 0, 2, 1e-4, 5)

Method did not converge. The last iteration gives  1.3125  with
function value  -0.14562511444091797
```

Theorem 28. *Suppose that $f \in C^0[a, b]$ and $f(a)f(b) < 0$. The bisection method generates a sequence $\{p_n\}$ approximating a zero p of $f(x)$ with*

$$|p_n - p| \leq \frac{b-a}{2^n}, n \geq 1.$$

Proof. Let the sequences $\{a_n\}$ and $\{b_n\}$ denote the left-end and right-end points of the subintervals generated by the bisection method. Since at each step the interval is halved, we have

$$b_n - a_n = \frac{1}{2}(b_{n-1} - a_{n-1}).$$

By mathematical induction, we get

$$b_n - a_n = \frac{1}{2}(b_{n-1} - a_{n-1}) = \frac{1}{2^2}(b_{n-2} - a_{n-2}) = ... = \frac{1}{2^{n-1}}(b_1 - a_1).$$

Therefore $b_n - a_n = \frac{1}{2^{n-1}}(b-a)$. Observe that

$$|p_n - p| \leq \frac{1}{2}(b_n - a_n) = \frac{1}{2^n}(b-a) \tag{2.3}$$

and thus $|p_n - p| \to 0$ as $n \to \infty$. □

Corollary 29. *The bisection method has linear convergence.*

Proof. The bisection method does not satisfy (2.1) for any $C < 1$, but it satisfies a variant of (2.2) with $C = 1/2$ from the previous theorem. □

Finding the number of iterations to obtain a specified accuracy: Can we find n that will ensure $|p_n - p| \leq 10^{-L}$ for some given L?

We have, from the proof of the previous theorem (see (2.3)) : $|p_n - p| \leq \frac{1}{2^n}(b-a)$. Therefore, we can make $|p_n - p| \leq 10^{-L}$, by choosing n large enough so that the upper bound $\frac{1}{2^n}(b-a)$ is less than 10^{-L} :

$$\frac{1}{2^n}(b-a) \leq 10^{-L} \Rightarrow n \geq \log_2\left(\frac{b-a}{10^{-L}}\right).$$

Example 30. Determine the number of iterations necessary to solve $f(x) = x^5 + 2x^3 - 5x - 2 = 0$ with accuracy $10^{-4}, a = 0, b = 2$.

Solution. Since $n \geq \log_2\left(\frac{2}{10^{-4}}\right) = 4\log_2 10 + 1 = 14.3$, the number of required iterations is 15.

Exercise 2.2-1: Find the root of $f(x) = x^3 + 4x^2 - 10$ using the bisection method, with the following specifications:

a) Modify the Python code for the bisection method so that the only stopping criterion is whether $f(p) = 0$ (remove the other criterion from the code). Also, add a print statement to the code, so that every time a new p is computed, Python prints the value of p and the iteration number.

b) Find the number of iterations N necessary to obtain an accuracy of 10^{-4} for the root, using the theoretical results of Section 2.2. (The function $f(x)$ has one real root in $(1, 2)$, so set $a = 1, b = 2$.)

c) Run the code using the value for N obtained in part (b) to compute $p_1, p_2, ..., p_N$ (set $a = 1, b = 2$ in the modified Python code).

d) The actual root, correct to six digits, is $p = 1.36523$. Find the absolute error when p_N is used to approximate the actual root, that is, find $|p - p_N|$. Compare this error, with the upper bound for the error used in part (b).

Exercise 2.2-2: Find an approximation to $25^{1/3}$ correct to within 10^{-5} using the bisection algorithm, following the steps below:

a) First express the problem as $f(x) = 0$ with $p = 25^{1/3}$ the root.

b) Find an interval (a, b) that contains the root, using Intermediate Value Theorem.

c) Determine, analytically, the number of iterates necessary to obtain the accuracy of 10^{-5}.

d) Use the Python code for the bisection method to compute the iterate from (c), and compare the actual absolute error with 10^{-5}.

2.3 Newton's method

Suppose $f \in C^2[a, b]$, i.e., f, f', f'' are continuous on $[a, b]$. Let p_0 be a "good" approximation to p such that $f'(p_0) \neq 0$ and $|p - p_0|$ is "small". First Taylor polynomial for f at p_0 with the remainder term is

$$f(x) = f(p_0) + (x - p_0)f'(p_0) + \frac{(x - p_0)^2}{2!}f''(\xi(x))$$

where $\xi(x)$ is a number between x and p_0. Substitute $x = p$ and note $f(p) = 0$ to get:

$$0 = f(p_0) + (p - p_0)f'(p_0) + \frac{(p - p_0)^2}{2!}f''(\xi(p))$$

where $\xi(p)$ is a number between p and p_0. Rearrange the equation to get

$$p = p_0 - \frac{f(p_0)}{f'(p_0)} - \frac{(p - p_0)^2}{2}\frac{f''(\xi(p))}{f'(p_0)}. \tag{2.4}$$

If $|p - p_0|$ is "small" then $(p - p_0)^2$ is even smaller, and the error term can be dropped to obtain the following approximation:

$$p \approx p_0 - \frac{f(p_0)}{f'(p_0)}.$$

The idea in Newton's method is to set the next iterate, p_1, to this approximation:

$$p_1 = p_0 - \frac{f(p_0)}{f'(p_0)}.$$

Equation (2.4) can be written as

$$p = p_1 - \frac{(p-p_0)^2}{2}\frac{f''(\xi(p))}{f'(p_0)}. \tag{2.5}$$

Summary: Start with an initial approximation p_0 to p and generate the sequence $\{p_n\}_{n=1}^{\infty}$ by

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, n \geq 1. \tag{2.6}$$

This is called Newton's method.

Graphical interpretation:

Start with p_0. Draw the tangent line at $(p_0, f(p_0))$ and approximate p by the intercept p_1 of the line:

$$f'(p_0) = \frac{0 - f(p_0)}{p_1 - p_0} \Rightarrow p_1 - p_0 = -\frac{f(p_0)}{f'(p_0)} \Rightarrow p_1 = p_0 - \frac{f(p_0)}{f'(p_0)}.$$

Now draw the tangent at $(p_1, f(p_1))$ and continue.

Remark 31.

1. Clearly Newton's method will fail if $f'(p_n) = 0$ for some n. Graphically this means the tangent line is parallel to the x-axis so we cannot get the x-intercept.

2. Newton's method may fail to converge if the initial guess p_0 is not close to p. In Figure (2.3), either choice for p_0 results in a sequence that oscillates between two points.

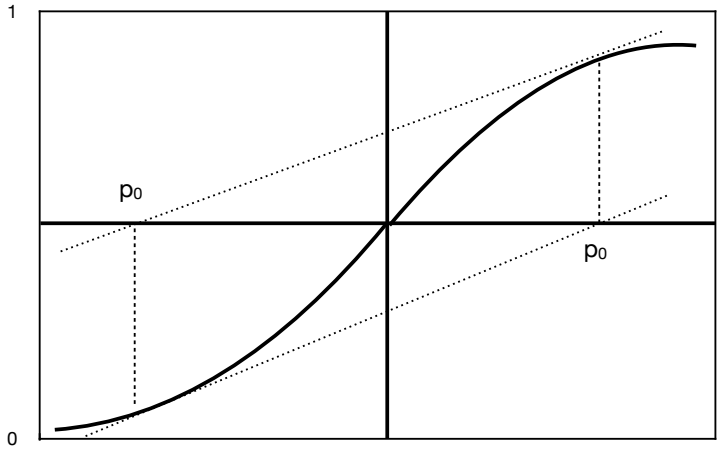


Figure 2.3: Non-converging behavior for Newton's method

3. Newton's method requires $f'(x)$ is known explicitly.

Exercise 2.3-1: Sketch the graph for $f(x) = x^2 - 1$. What are the roots of the equation $f(x) = 0$?

1. Let $p_0 = 1/2$ and find the first two iterations p_1, p_2 of Newton's method by hand. Mark the iterates on the graph of f you sketched. Do you think the iterates will converge to a zero of f?

2. Let $p_0 = 0$ and find p_1. What are your conclusions about the convergence of the iterates?

Python code for Newton's method

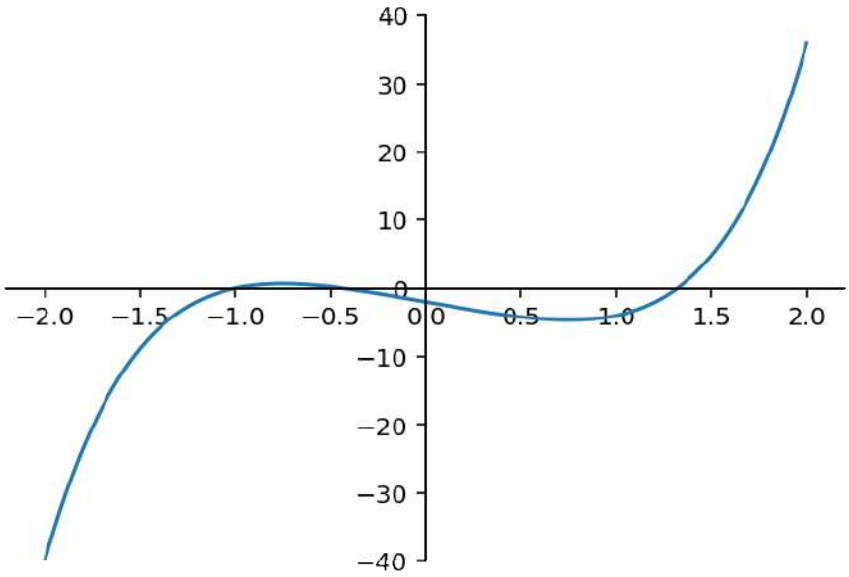
The Python code below is based on Equation (2.6). The variable *pin* in the code corresponds to p_{n-1}, and p corresponds to p_n. The code overwrites these variables as the iteration continues. Also notice that the code has two functions as inputs; f and *fprime* (the derivative f').

```
In [1]: def newton(f, fprime, pin, eps, N):
            n = 1
            p = 0. # to ensure the value of p carries out of the while loop
            while n <= N:
                p = pin - f(pin)/fprime(pin)
                if np.isclose(f(p), 0) or np.abs(p-pin) < eps:
                    print('p is ', p, ' and the iteration number is ', n)
                    return
                pin = p
                n += 1
            y = f(p)
            print('Method did not converge. The last iteration gives ',
                  p, ' with function value ', y)
```

Let's apply Newton's method to find the root of $f(x) = x^5 + 2x^3 - 5x - 2$, a function we considered before. First, we plot the function.

```
In [2]: import matplotlib.pyplot as plt
        %matplotlib inline
        import numpy as np

In [3]: x = np.linspace(-2, 2, 1000)
        y = x**5+2*x**3-5*x-2
        ax = plt.gca()
        ax.spines['left'].set_position('center')
```

```
ax.spines['right'].set_position('center')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_position('center')
ax.set_ylim([-40, 40])
plt.plot(x,y);
```



The derivative is $f' = 5x^4 + 6x^2 - 5$, we set pin $= 1$, eps $= \epsilon = 10^{-4}$, and $N = 20$, in the code.

```
In [4]: newton(lambda x: x**5+2*x**3-5*x-2, lambda x: 5*x**4+6*x**2-5, 1, 1e-4, 20)

p is  1.3196411672093726  and the iteration number is  6
```

Recall that the bisection method required 16 iterations to approximate the root in $[0, 2]$ as $p = 1.31967$. (However, the stopping criterion used in bisection and Newton's methods are slightly different.) 1.3196 is the rightmost root in the plot. But there are other roots of the function. Let's run the code with pin $= 0$.

```
In [5]: newton(lambda x: x**5+2*x**3-5*x-2, lambda x: 5*x**4+6*x**2-5, 0, 1e-4, 20)

p is  -0.43641313299799755  and the iteration number is  4
```

Now we use pin $= -2.0$ which will give the leftmost root.

```
In [6]: newton(lambda x: x**5+2*x**3-5*x-2, lambda x: 5*x**4+6*x**2-5, -2, 1e-4, 20)

p is  -1.0000000001014682  and the iteration number is  7
```

Theorem 32. *Let $f \in C^2[a,b]$ and assume $f(p) = 0, f'(p) \neq 0$ for $p \in (a,b)$. If p_0 is chosen sufficiently close to p, then Newton's method generates a sequence that converges to p with*

$$\lim_{n\to\infty} \frac{p - p_{n+1}}{(p-p_n)^2} = -\frac{f''(p)}{2f'(p)}.$$

Proof. Since f' is continuous and $f'(p) \neq 0$, there exists an interval $I = [p-\epsilon, p+\epsilon]$ on which $f' \neq 0$. Let

$$M = \frac{\max_{x\in I} |f''(x)|}{2\min_{x\in I} |f'(x)|}.$$

Pick p_0 from the interval I (which means $|p-p_0| \leq \epsilon$), sufficiently close to p so that $M|p-p_0| < 1$. From Equation (2.5) we have:

$$|p-p_1| = \frac{|p-p_0||p-p_0|}{2}\left|\frac{f''(\xi(p))}{f'(p_0)}\right| < |p-p_0||p-p_0|M < |p-p_0| \leq \epsilon. \tag{2.7}$$

Multiply both sides of $|p-p_1| < |p-p_0|$ by M to get $M|p-p_1| < M|p-p_0| < 1$. Therefore, we have obtained: $|p-p_1| < \epsilon$ and $M|p-p_1| < 1$. Repeating the same argument used in 2.7 to $|p-p_2|$, we can show $|p-p_2| < \epsilon$ and $M|p-p_2| < 1$. Therefore by induction $|p-p_n| < \epsilon$ and $M|p-p_n| < 1$, for all n. This implies that all the iterates p_n are in the interval I so $f'(p_n)$ is never zero in Newton's iteration.

If we replace p_1 by p_{n+1}, and p_0 by p_n in Equation (2.5), we get

$$p - p_{n+1} = -\frac{(p-p_n)^2}{2}\frac{f''(\xi(p))}{f'(p_n)}. \tag{2.8}$$

Here $\xi(p)$ is a number between p and p_n. Since $\xi(p)$ changes recursively with n, let's update our notation as: $\xi(p) = \xi_n$. Then, Equation (2.8) implies

$$|p-p_{n+1}| \leq M|p-p_n|^2 \Rightarrow M|p-p_{n+1}| \leq (M|p-p_n|)^2.$$

Similarly, $|p-p_n| \leq M|p-p_{n-1}|^2$, or $M|p-p_n| \leq (M|p-p_{n-1}|)^2$, and thus $M|p-p_{n+1}| \leq (M|p-p_{n-1}|)^{2^2}$. By induction, we can show

$$M|p-p_n| \leq (M|p-p_0|)^{2^n} \Rightarrow |p-p_n| \leq \frac{1}{M}(M|p-p_0|)^{2^n}.$$

Since $M|p-p_0| < 1$, $|p-p_n| \to 0$ as $n \to \infty$. Therefore $\lim_{n\to\infty} p_n = p$. Finally,

$$\lim_{n\to\infty} \frac{p-p_{n+1}}{(p-p_n)^2} = \lim_{n\to\infty} -\frac{1}{2}\frac{f''(\xi_n)}{f'(p_n)},$$

and since $p_n \to p$, and ξ_n is between p_n and p, $\xi_n \to p$, and therefore

$$\lim_{n\to\infty} \frac{p-p_{n+1}}{(p-p_n)^2} = -\frac{1}{2}\frac{f''(p)}{f'(p)}$$

proving the theorem. □

Corollary 33. *Newton's method has quadratic convergence.*

Proof. Recall that quadratic convergence means

$$|p_{n+1}-p| \leq C|p_n-p|^2,$$

for some constant $C > 0$. Taking the absolute values of the limit established in the previous theorem, we obtain

$$\lim_{n\to\infty} \left|\frac{p-p_{n+1}}{(p-p_n)^2}\right| = \lim_{n\to\infty} \frac{|p_{n+1}-p|}{|p_n-p|^2} = \left|\frac{1}{2}\frac{f''(p)}{f'(p)}\right|.$$

Let $C' = \left|\frac{1}{2}\frac{f''(p)}{f'(p)}\right|$. From the definition of limit of a sequence, for any $\epsilon > 0$, there exists an integer $N > 0$ such that $\frac{|p_{n+1}-p|}{|p_n-p|^2} < C' + \epsilon$ whenever $n > N$. Set $C = C' + \epsilon$ to obtain $|p_{n+1}-p| \leq C|p_n-p|^2$ for $n > N$. □

Example 34. The Black-Scholes-Merton (BSM) formula, for which Myron Scholes and Robert Merton were awarded the Nobel prize in economics in 1997, computes the fair price of a contract known as the **European call option**. This contract gives its owner the right to purchase the asset the contract is written on (for example, a stock), for a specific price denoted by K and called the strike price (or exercise price), at a future time denoted by T and called the expiry. The formula gives the value of the European call option, C, as

$$C = S\phi(\mathrm{d}_{\mathbb{1}}) - Ke^{-rT}\phi(\mathrm{d}_{\mathbb{2}})$$

where S is the price of the asset at the present time, r is the risk-free interest rate, and $\phi(x)$ is the distribution function of the standard normal random variable, given by

$$\phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-t^2/2}dt.$$

The constants $\mathrm{d}_{\mathbb{1}}, \mathrm{d}_{\mathbb{2}}$ are obtained from

$$\mathrm{d}_{\mathbb{1}} = \frac{\log(S/K) + (r+\sigma^2/2)T}{\sigma\sqrt{T}},\ \mathrm{d}_{\mathbb{2}} = \mathrm{d}_{\mathbb{1}} - \sigma\sqrt{T}.$$

All the constants in the BSM formula can be observed, except for σ, which is called the volatility of the underlying asset. It has to be estimated from empirical data in some way. We want to concentrate on the relationship between C and σ, and think of C as a function of σ only. We rewrite the BSM formula emphasizing σ:

$$C(\sigma) = S\phi(\mathrm{d}_1) - Ke^{-rT}\phi(\mathrm{d}_2)$$

It may look like the independent variable σ is missing on the right hand side of the above formula, but it is not: the constants $\mathrm{d}_1, \mathrm{d}_2$ both depend on σ. We can also think about $\mathrm{d}_1, \mathrm{d}_2$ as functions of σ.

There are two questions financial engineers are interested in:

- Compute the option price C based on an estimate of σ
- Observe the price of an option $\hat{C}$ traded in the market, and find σ^* for which the BSM formula gives the output $\hat{C}$, i.e, $C(\sigma^*) = \hat{C}$. The volatility σ^* obtained in this way is called the **implied volatility**.

The second question can be answered using a root-finding method, in particular, Newton's method. To summarize, we want to solve the equation:

$$C(\sigma) - \hat{C} = 0$$

where $\hat{C}$ is a given constant, and

$$C(\sigma) = S\phi(\mathrm{d}_1) - Ke^{-rT}\phi(\mathrm{d}_2).$$

To use Newton's method, we need $C'(\sigma) = \frac{dC}{d\sigma}$. Since $\mathrm{d}_1, \mathrm{d}_2$ are functions of σ, we have

$$\frac{dC}{d\sigma} = S\frac{d\phi(\mathrm{d}_1)}{d\sigma} - Ke^{-rT}\frac{d\phi(\mathrm{d}_2)}{d\sigma}. \tag{2.9}$$

Let's compute the derivatives on the right hand side of (2.9).

$$\frac{d\phi(\mathrm{d}_1)}{d\sigma} = \frac{d}{d\sigma}\left(\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\mathrm{d}_1} e^{-t^2/2}dt\right) = \frac{1}{\sqrt{2\pi}}\left(\frac{d}{d\sigma}\underbrace{\int_{-\infty}^{\mathrm{d}_1} e^{-t^2/2}dt}_{u}\right).$$

We will use the chain rule to compute the derivative $\frac{d}{d\sigma}\underbrace{\int_{-\infty}^{\mathrm{d}_1} e^{-t^2/2}dt}_{u} = \frac{du}{d\sigma}$:

$$\frac{du}{d\sigma} = \frac{du}{d\mathrm{d}_1}\frac{d\mathrm{d}_1}{d\sigma}.$$

The first derivative follows from the Fundamental Theorem of Calculus

$$\frac{du}{d\mathbb{d}_1} = e^{-\mathbb{d}_1{}^2/2},$$

and the second derivative is an application of the quotient rule of differentiation

$$\frac{d\mathbb{d}_1}{d\sigma} = \frac{d}{d\sigma}\left(\frac{\log(S/K) + (r+\sigma^2/2)T}{\sigma\sqrt{T}}\right) = \sqrt{T} - \frac{\log(S/K) + (r+\sigma^2/2)T}{\sigma^2\sqrt{T}}.$$

Putting the pieces together, we have

$$\frac{d\phi(\mathbb{d}_1)}{d\sigma} = \frac{e^{-\mathbb{d}_1{}^2/2}}{\sqrt{2\pi}}\left(\sqrt{T} - \frac{\log(S/K) + (r+\sigma^2/2)T}{\sigma^2\sqrt{T}}\right).$$

Going back to the second derivative we need to compute in equation (2.9), we have:

$$\frac{d\phi(\mathbb{d}_2)}{d\sigma} = \frac{1}{\sqrt{2\pi}}\left(\frac{d}{d\sigma}\int_{-\infty}^{\mathbb{d}_2} e^{-t^2/2}dt\right).$$

Using the chain rule and the Fundamental Theorem of Calculus we obtain

$$\frac{d\phi(\mathbb{d}_2)}{d\sigma} = \frac{e^{-\mathbb{d}_2{}^2/2}}{\sqrt{2\pi}}\frac{d\mathbb{d}_2}{d\sigma}.$$

Since $\mathbb{d}_2$ is defined as $\mathbb{d}_2 = \mathbb{d}_1 - \sigma\sqrt{T}$, we can express $d\mathbb{d}_2/d\sigma$ in terms of $d\mathbb{d}_1/d\sigma$ as:

$$\frac{d\mathbb{d}_2}{d\sigma} = \frac{d\mathbb{d}_1}{d\sigma} - \sqrt{T}.$$

Finally, we have the derivative we need:

$$\frac{dC}{d\sigma} = \frac{Se^{-\frac{\mathbb{d}_1{}^2}{2}}}{\sqrt{2\pi}}\left(\sqrt{T} - \frac{\log(\frac{S}{K}) + (r+\frac{\sigma^2}{2})T}{\sigma^2\sqrt{T}}\right) + K\frac{e^{-(rT+\frac{\mathbb{d}_2{}^2}{2})}}{\sqrt{2\pi}}\left(\frac{\log(\frac{S}{K}) + (r+\frac{\sigma^2}{2})T}{\sigma^2\sqrt{T}}\right) \tag{2.10}$$

We are ready to apply Newton's method to solve the equation $C(\sigma) - \hat{C} = 0$. Now let's find some data.

The General Electric Company (GE) stock is \$7.01 on Dec 8, 2018, and a European call option on this stock, expiring on Dec 14, 2018, is priced at \$0.10. The option has strike price $K = \$7.5$. The risk-free interest rate is 2.25%. The expiry T is measured in years, and since there are 252 trading days in a year, $T = 6/252$. We put this information in Python:

```
In [1]: S = 7.01
        K = 7.5
        r = 0.0225
        T = 6/252
```

We have not discussed how to compute the distribution function of the standard normal random variable $\phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$. In Chapter 4, we will discuss how to compute integrals numerically, but for this example, we will use the built-in function Python has for $\phi(x)$. It is in a subpackage of SciPy called **stats**:

```
In [2]: from scipy import stats
```

The following defines **stdnormal** as the standard normal random variable.

```
In [3]: stdnormal = stats.norm(loc=0, scale=1)
```

The member function **cdf**(x) of **stdnormal** computes the standard normal distribution function at x. We write a function **phi(x)** based on this member function, matching our notation $\phi(x)$ for the distribution function.

```
In [4]: phi = lambda x: stdnormal.cdf(x)
```

Next we define $C(\sigma)$ and $C'(\sigma)$. In the Python code, we replace σ by x.

```
In [5]: def c(x):
            d1 = (np.log(S/K)+(r+x**2/2)*T) / (x*np.sqrt(T))
            d2 = d1 - x*np.sqrt(T)
            return S*phi(d1) - K*np.exp(-r*T)*phi(d2)
```

The function **cprime**(x) is based on equation (2.10):

```
In [6]: def cprime(x):
            d1 = (np.log(S/K)+(r+x**2/2)*T) / (x*np.sqrt(T))
            d2 = d1 - x*np.sqrt(T)
            A = (np.log(S/K)+(r+x**2/2)*T) / (np.sqrt(T)*x**2)
            return S*(np.exp(-d1**2/2) / np.sqrt(2*np.pi)) * (np.sqrt(T)-A) \
            + K*np.exp(-(r*T+d2**2/2)) * A / np.sqrt(2*np.pi)
```

We then load the **newton** function and run it to find the implied volatility which turns out to be 62%.

```
In [7]: newton(lambda x: c(x)-0.1, cprime, 1, 1e-4, 60)

p is  0.6231138483741047  and the iteration number is  3
```

2.4 Secant method

One drawback of Newton's method is that we need to know $f'(x)$ explicitly to evaluate $f'(p_{n-1})$ in

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, n \geq 1.$$

If we do not know $f'(x)$ explicitly, or if its computation is expensive, we might approximate $f'(p_{n-1})$ by the finite difference

$$\frac{f(p_{n-1}+h) - f(p_{n-1})}{h} \tag{2.11}$$

for some small h. We then need to compute two values of f at each iteration to approximate f'. Determining h in this formula brings some difficulty, but there is a way to get around this. We will use the iterates themselves to rewrite the finite difference (2.11) as

$$\frac{f(p_{n-1}) - f(p_{n-2})}{p_{n-1} - p_{n-2}}.$$

Then, the recursion for p_n simplifies as

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{\frac{f(p_{n-1})-f(p_{n-2})}{p_{n-1}-p_{n-2}}} = p_{n-1} - f(p_{n-1})\frac{p_{n-1}-p_{n-2}}{f(p_{n-1})-f(p_{n-2})}, n \geq 2. \tag{2.12}$$

This is called the secant method. Observe that

1. No additional function evaluations are needed,
2. The recursion requires two initial guesses p_0, p_1.

Geometric interpretation: The slope of the secant line through the points $(p_{n-1}, f(p_{n-1}))$ and $(p_{n-2}, f(p_{n-2}))$ is $\frac{f(p_{n-1})-f(p_{n-2})}{p_{n-1}-p_{n-2}}$. The x-intercept of the secant line, which is set to p_n, is

$$\frac{0 - f(p_{n-1})}{p_n - p_{n-1}} = \frac{f(p_{n-1}) - f(p_{n-2})}{p_{n-1} - p_{n-2}} \Rightarrow p_n = p_{n-1} - f(p_{n-1})\frac{p_{n-1}-p_{n-2}}{f(p_{n-1})-f(p_{n-2})}$$

which is the recursion of the secant method.

The following theorem shows that if the initial guesses are "good", the secant method has superlinear convergence. A proof can be found in Atkinson [3].

Theorem 35. *Let $f \in C^2[a,b]$ and assume $f(p) = 0, f'(p) \neq 0$, for $p \in (a,b)$. If the initial guesses p_0, p_1 are sufficiently close to p, then the iterates of the secant method converge to p with*

$$\lim_{n\to\infty} \frac{|p - p_{n+1}|}{|p - p_n|^{r_0}} = \left|\frac{f''(p)}{2f'(p)}\right|^{r_1}$$

where $r_0 = \frac{\sqrt{5}+1}{2} \approx 1.62, r_1 = \frac{\sqrt{5}-1}{2} \approx 0.62$.

Python code for the secant method

The following code is based on Equation (2.12); the recursion for the secant method. The initial guesses are called *pzero* and *pone* in the code. The same stopping criterion as in Newton's method is used. Notice that once a new iterate p is computed, *pone* is updated as p, and *pzero* is updated as *pone*.

```
In [1]: def secant(f, pzero, pone, eps, N):
            n = 1
            p = 0. # to ensre the value of p carries out of the while loop
            while n <= N:
                p = pone - f(pone)*(pone-pzero) / (f(pone)-f(pzero))
                if np.isclose(f(p), 0) or np.abs(p-pone)<eps:
                    print('p is ', p, ' and the iteration number is ', n)
                    return
                pzero = pone
                pone = p
                n += 1
            y = f(p)
            print('Method did not converge. The last iteration gives ',
                  p, ' with function value ', y)
```

Let's find the root of $f(x) = \cos x - x$ using the secant method, using 0.5 and 1 as the initial guesses.

```
In [2]: secant(lambda x: np.cos(x)-x, 0.5, 1, 1e-4, 20)

p is  0.739085132900112  and the iteration number is  4
```

Exercise 2.4-1: Use the Python codes for the secant and Newton's methods to find solutions for the equation $\sin x - e^{-x} = 0$ on $0 \leq x \leq 1$. Set tolerance to 10^{-4}, and take $p_0 = 0$ in Newton, and $p_0 = 0, p_1 = 1$ in secant method. Do a visual inspection of the estimates and comment on the convergence rates of the methods.

Exercise 2.4-2:

a) The function $y = \log x$ has a root at $x = 1$. Run the Python code for Newton's method with $p_0 = 2, \epsilon = 10^{-4}, N = 20$, and then try $p_0 = 3$. Does Newton's method find the root in each case? If Python gives an error message, explain what the error is.

b) One can combine the bisection method and Newton's method to develop a hybrid method that converges for a wider range of starting values p_0, and has better convergence rate than the bisection method.

Write a Python code for a bisection-Newton hybrid method, as described below. (You can use the Python codes for the bisection and Newton's methods from the lecture notes.) Your code will input f, f', a, b, ϵ, N where f, f' are the function and its derivative, (a, b) is an interval that contains the root (i.e., $f(a)f(b) < 0$), and ϵ, N are the tolerance and the maximum number of iterations. The code will use the same stopping criterion used in Newton's method.

The method will start with computing the midpoint of (a, b), call it p_0, and use Newton's method with initial guess p_0 to obtain p_1. It will then check whether $p_1 \in (a, b)$. If $p_1 \in (a, b)$, then the code will continue using Newton's method to compute the next iteration p_2. If $p_1 \notin (a, b)$, then we will not accept p_1 as the next iteration: instead the code will switch to the bisection method, determine which subinterval among $(a, p_0), (p_0, b)$ contains the root, updates the interval (a, b) as the subinterval that contains the root, and sets p_1 to the midpoint of this interval. Once p_1 is obtained, the code will check if the stopping criterion is satisfied. If it is satisfied, the code will return p_1 and the iteration number, and terminate. If it is not satisfied, the code will use Newton's method, with p_1 as the initial guess, to compute p_2. Then it will check whether $p_2 \in (a, b)$, and continue in this way. If the code does not terminate after N iterations, output an error message similar to Newton's method.

Apply the hybrid method to:

- a polynomial with a known root, and check if the method finds the correct root;
- $y = \log x$ with $(a, b) = (0, 6)$, for which Newton's method failed in part (a).

c) Do you think in general the hybrid method converges to the root, provided the initial interval (a, b) contains the root, for any starting value p_0? Explain.

2.5 Muller's method

The secant method uses a linear function that passes through $(p_0, f(p_0))$ and $(p_1, f(p_1))$ to find the next iterate p_2. Muller's method takes three initial approximations, passes a parabola (quadratic polynomial) through $(p_0, f(p_0))$, $(p_1, f(p_1))$, $(p_2, f(p_2))$, and uses **one** of the roots of the polynomial as the next iterate.

Let the quadratic polynomial written in the following form

$$P(x) = a(x - p_2)^2 + b(x - p_2) + c. \tag{2.13}$$

Solve the following equations for a, b, c

$$\begin{aligned} P(p_0) &= f(p_0) = a(p_0 - p_2)^2 + b(p_0 - p_2) + c \\ P(p_1) &= f(p_1) = a(p_1 - p_2)^2 + b(p_1 - p_2) + c \\ P(p_2) &= f(p_2) = c \end{aligned}$$

to get

$$\begin{aligned} c &= f(p_2) \\ b &= \frac{(p_0 - p_2)(f(p_1) - f(p_2))}{(p_1 - p_2)(p_0 - p_1)} - \frac{(p_1 - p_2)(f(p_0) - f(p_2))}{(p_0 - p_2)(p_0 - p_1)} \\ a &= \frac{f(p_0) - f(p_2)}{(p_0 - p_2)(p_0 - p_1)} - \frac{f(p_1) - f(p_2)}{(p_1 - p_2)(p_0 - p_1)}. \end{aligned} \tag{2.14}$$

Now that we have determined $P(x)$, the next step is to solve $P(x) = 0$, and set the next iterate p_3 to its solution. To this end, put $w = x - p_2$ in (2.13) to rewrite the quadratic equation as

$$aw^2 + bw + c = 0.$$

From the quadratic formula, we obtain the roots

$$\hat{w} = \hat{x} - p_2 = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}. \tag{2.15}$$

Let $\Delta = b^2 - 4ac$. We have two roots (which could be complex numbers), $-2c/(b + \sqrt{\Delta})$ and $-2c/(b - \sqrt{\Delta})$, and we need to pick one of them. We will pick the root that is closer to p_2, in other words, the root that makes $|\hat{x} - p_2|$ the smallest. (If the numbers are complex, the absolute value means the norm of the complex number.) Therefore we have

$$\hat{x} - p_2 = \begin{cases} \frac{-2c}{b+\sqrt{\Delta}} & \text{if } |b + \sqrt{\Delta}| > |b - \sqrt{\Delta}| \\ \frac{-2c}{b-\sqrt{\Delta}} & \text{if } |b + \sqrt{\Delta}| \leq |b - \sqrt{\Delta}| \end{cases}. \tag{2.16}$$

The next iterate of Muller's method, p_3, is set to the value of $\hat{x}$ obtained from the above calculation, that is,

$$p_3 = \hat{x} = \begin{cases} p_2 - \frac{2c}{b+\sqrt{\Delta}} & \text{if } |b + \sqrt{\Delta}| > |b - \sqrt{\Delta}| \\ p_2 - \frac{2c}{b-\sqrt{\Delta}} & \text{if } |b + \sqrt{\Delta}| \leq |b - \sqrt{\Delta}| \end{cases}.$$

Remark 36.

1. Muller's method can find real as well as complex roots.

2. The convergence of Muller's method is superlinear, that is,

$$\lim_{n\to\infty} \frac{|p - p_{n+1}|}{|p - p_n|^{\alpha}} = \left| \frac{f^{(3)}(p)}{6f'(p)} \right|^{\frac{\alpha-1}{2}}$$

where $\alpha \approx 1.84$, provided $f \in C^3[a, b], p \in (a, b)$, and $f'(p) \neq 0$.

3. Muller's method converges for a variety of starting values even though pathological examples that do not yield convergence can be found (for example, when the three starting values fall on a line).

Python code for Muller's method

The following Python code takes initial guesses p_0, p_1, p_2 (written as *pzero, pone, ptwo* in the code), computes the coefficients a, b, c from Equation (2.14), and sets the root p_3 to p. It then updates the three initial guesses as the last three iterates, and continues until the stopping criterion is satisfied.

We need to compute the square root, and the absolute value, of possibly complex numbers in Equations (2.15) and (2.16). The Python function for the square root of a possibly complex number z is **complex(z)**$^{0.5}$, and its absolute value is **np.abs(z)**.

```
In [1]: def muller(f, pzero, pone, ptwo, eps, N):
            n = 1
            p = 0
            while n <= N:
                c = f(ptwo)
                b1 = (pzero-ptwo) * (f(pone)-f(ptwo)) / ((pone-ptwo)*(pzero-pone))
                b2 = (pone-ptwo) * (f(pzero)-f(ptwo)) / ((pzero-ptwo)*(pzero-pone))
                b = b1 - b2
                a1 = (f(pzero)-f(ptwo)) / ((pzero-ptwo)*(pzero-pone))
                a2 = (f(pone)-f(ptwo)) / ((pone-ptwo)*(pzero-pone))
                a = a1 - a2
                d = (complex(b**2-4*a*c))**0.5
                if np.abs(b-d) < np.abs(b+d):
                    inc = 2*c/(b+d)
                else:
                    inc = 2*c/(b-d)
                p = ptwo - inc
                if np.isclose(f(p), 0) or np.abs(p-ptwo)<eps:
                    print('p is ', p, ' and the iteration number is ', n)
                    return
```

```
            pzero = pone
            pone = ptwo
            ptwo = p
            n += 1
        y = f(p)
        print('Method did not converge. The last iteration gives ',
              p, ' with function value ', y)
```

The polynomial x^5+2x^3-5x-2 has three real roots, and two complex roots that are conjugates. Let's find them all, by experimenting with various initial guesses.

```
In [2]: muller(lambda x: x**5+2*x**3-5*x-2, 0.5, 1.0, 1.5, 1e-5, 10)

p is  (1.3196411677283386+0j)  and the iteration number is  4


In [3]: muller(lambda x: x**5+2*x**3-5*x-2, 0.5, 0, -0.1, 1e-5, 10)

p is  (-0.43641313299908585+0j)  and the iteration number is  5


In [4]: muller(lambda x: x**5+2*x**3-5*x-2, 0, -0.1, -1, 1e-5, 10)

p is  (-1+0j)  and the iteration number is  1


In [5]: muller(lambda x: x**5+2*x**3-5*x-2, 5, 10, 15, 1e-5, 20)

p is  (0.05838598289491982+1.8626227582154478j)  and the iteration number is  18
```

2.6 Fixed-point iteration

Many root-finding methods are based on the so-called fixed-point iteration, a method we discuss in this section.

Definition 37. A number p is a fixed-point for a function $g(x)$ if $g(p) = p$.

We have two problems that are related to each other:

- **Fixed-point problem**: Find p such that $g(p) = p$.
- **Root-finding problem**: Find p such that $f(p) = 0$.

We can formulate a root-finding problem as a fixed-point problem, and vice versa. For example, assume we want to solve the root finding problem, $f(p) = 0$. Define $g(x) = x - f(x)$, and observe that if p is a fixed-point of $g(x)$, that is, $g(p) = p - f(p) = p$, then p is a root of $f(x)$. Here the function g is not unique: there are many ways one can represent the root-finding problem $f(p) = 0$ as a fixed-point problem, and as we will learn later, not all will be useful to us in developing fixed-point iteration algorithms.

The next theorem answers the following questions: When does a function g have a fixed-point? If it has a fixed-point, is it unique?

Theorem 38. *1. If g is a continuous function on $[a, b]$ and $g(x) \in [a, b]$ for all $x \in [a, b]$, then g has at least one fixed-point in $[a, b]$.*

2. If, in addition, $|g(x) - g(y)| \leq \lambda|x - y|$ for all $x, y \in [a, b]$ where $0 < \lambda < 1$, then the fixed-point is unique.

Proof. Consider $f(x) = g(x) - x$. Assume $g(a) \neq a$ and $g(b) \neq b$ (otherwise the proof is over.) Then $f(a) = g(a) - a > 0$ since $g(a)$ must be greater than a if it's not equal to a. Similarly, $f(b) = g(b) - b < 0$. Then from IVT, there exists $p \in (a, b)$ such that $f(p) = 0$, or $g(p) = p$. To prove part 2, suppose there are two different fixed-points p, q. Then

$$|p - q| = |g(p) - g(q)| \leq \lambda|p - q| < |p - q|$$

which is a contradiction. □

Remark 39. Let g be a differentiable function on $[a, b]$ such that $|g'(x)| \leq k$ for all $x \in (a, b)$ for some positive constant $k < 1$. Then the hypothesis of part 2 of Theorem 38 is satisfied with $\lambda = k$. Indeed, from the mean value theorem

$$|g(x) - g(y)| = |g'(\xi)(x - y)| \leq k|x - y|$$

for all $x, y \in [a, b]$.

The following theorem describes how we can find a fixed point.

Theorem 40. *If g is a continuous function on $[a, b]$ satisfying the conditions*

1. $g(x) \in [a, b]$ for all $x \in [a, b]$,

2. $|g(x) - g(y)| \leq \lambda|x - y|$, for $x, y \in [a, b]$ where $0 < \lambda < 1$,

*then the **fixed-point iteration***

$$p_n = g(p_{n-1}), n \geq 1$$

converges to p, the unique fixed-point of g in $[a, b]$, for any starting point $p_0 \in [a, b]$.

Proof. Since $p_0 \in [a, b]$ and $g(x) \in [a, b]$ for all $x \in [a, b]$, all iterates $p_n \in [a, b]$. Observe that

$$|p - p_n| = |g(p) - g(p_{n-1})| \leq \lambda |p - p_{n-1}|.$$

Then by induction, $|p - p_n| \leq \lambda^n |p - p_0|$. Since $0 < \lambda < 1$, $\lambda^n \to 0$ as $n \to \infty$, and thus $p_n \to p$. □

Remark 41. Theorem 40 still holds if the second condition $|g(x) - g(y)| \leq \lambda |x - y|$, is replaced by $|g'(x)| \leq k$ for all $x \in [a, b]$ where $0 < k < 1$. (See Remark 39).

Corollary 42. *If g satisfies the hypothesis of Theorem 40, then the following error bounds hold.*

1. $|p - p_n| \leq \frac{\lambda^n}{1-\lambda} |p_1 - p_0|$
2. $|p - p_n| \leq \frac{1}{1-\lambda} |p_{n+1} - p_n|$
3. $|p - p_{n+1}| \leq \frac{\lambda}{1-\lambda} |p_{n+1} - p_n|$
4. $|p - p_n| \leq \lambda^n \max\{p_0 - a, b - p_0\}$

Geometric interpretation of fixed-point iteration

In Figures (2.4) and (2.5), take a starting value p_0 close to p, and mark the first few fixed-point iterations, p_0, p_1, p_2. Observe that the fixed-point iteration converges in the first graph, but diverges in the second one.

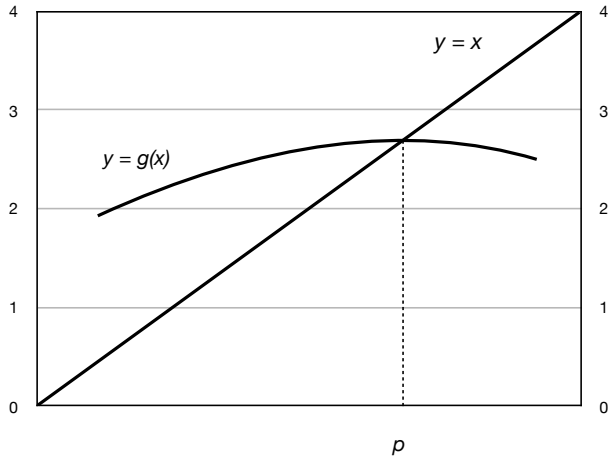


Figure 2.4: Fixed-point iteration: $|g'(p)| < 1$.

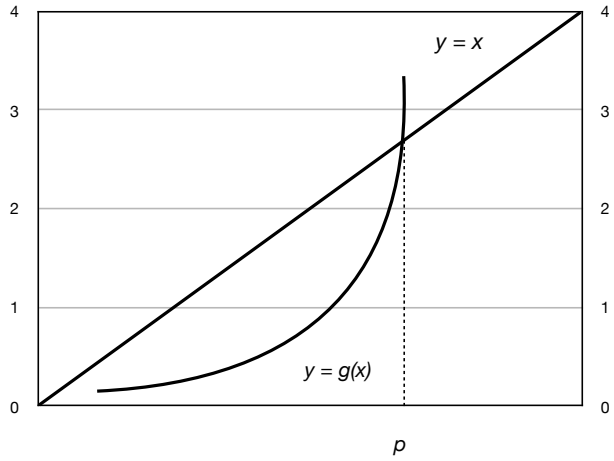


Figure 2.5: Fixed-point iteration: $|g'(p)| > 1$.

Example 43. Consider the root-finding problem $x^3 - 2x^2 - 1 = 0$ on $[1, 3]$.

1. Write the problem as a fixed-point problem, $g(x) = x$, for some g. Verify that the hypothesis of Theorem 40 (or Remark 41) is satisfied so that the fixed-point iteration converges.

2. Let $p_0 = 1$. Use Corollary 42 to find n that ensures an estimate to p accurate to within 10^{-4}.

Solution. 1. There are several ways we can write this problem as $g(x) = x$:

(a) Let $f(x) = x^3 - 2x^2 - 1$, and p be its root, that is, $f(p) = 0$. If we let $g(x) = x - f(x)$, then $g(p) = p - f(p) = p$, so p is a fixed-point of g. However, this choice for g will not be helpful, since g does not satisfy the first condition of Theorem 40: $g(x) \notin [1, 3]$ for all $x \in [1, 3]$ ($g(3) = -5 \notin [1, 3]$).

(b) Since p is a root for f, we have $p^3 = 2p^2 + 1$, or $p = (2p^2 + 1)^{1/3}$. Therefore, p is the solution to the fixed-point problem $g(x) = x$ where $g(x) = (2x^2 + 1)^{1/3}$.

- g is increasing on $[1, 3]$ and $g(1) = 1.44, g(3) = 2.67$, thus $g(x) \in [1, 3]$ for all $x \in [1, 3]$. Therefore, g satisfies the first condition of Theorem 40.
- $g'(x) = \frac{4x}{3(2x^2+1)^{2/3}}$ and $g'(1) = 0.64, g'(3) = 0.56$ and g' is decreasing on $[1, 3]$. Therefore g satisfies the condition in Remark 41 with $\lambda = 0.64$.

Then, from Theorem 40 and Remark 41, the fixed-point iteration converges if $g(x) = (2x^2 + 1)^{1/3}$.

2. Take $\lambda = k = 0.64$ in Corollary 42 and use bound (4):

$$|p - p_n| \leq (0.64)^n \max\{1 - 1, 3 - 1\} = 2(0.64^n).$$

We want $2(0.64^n) < 10^{-4}$, which implies $n \log 0.64 < -4 \log 10 - \log 2$, or $n > \frac{-4 \log 10 - \log 2}{\log 0.64} \approx 22.19$. Therefore $n = 23$ is the smallest number of iterations that ensures an absolute error of 10^{-4}.

Python code for fixed-point iteration

The following code starts with the initial guess p_0 (*pzero* in the code), computes $p_1 = g(p_0)$, and checks if the stopping criterion $|p_1 - p_0| < \epsilon$ is satisfied. If it is satisfied the code terminates with the value p_1. Otherwise p_1 is set to p_0, and the next iteration is computed.

```
In [1]: def fixedpt(g, pzero, eps, N):
            n = 1
            while n<N:
                pone = g(pzero)
                if np.abs(pone-pzero)<eps:
                    print('p is ', pone, ' and the iteration number is ', n)
                    return
                pzero = pone
                n += 1
            print('Did not converge. The last estimate is p = ', pzero)
```

Let's find the fixed-point of $g(x) = x$ where $g(x) = (2x^2 + 1)^{1/3}$, with $p_0 = 1$. We studied this problem in Example 43 where we found that 23 iterations guarantee an estimate accurate to within 10^{-4}. We set $\epsilon = 10^{-4}$, and $N = 30$, in the above code.

```
In [2]: fixedpt(lambda x: (2*x**2+1)**(1/3), 1, 1e-4, 30)
```

```
p is  2.205472095330031  and the iteration number is  19
```

The exact value of the fixed-point, equivalently the root of $x^3 - 2x^2 - 1$, is 2.20556943. Then the exact error is:

```
In [3]: 2.205472095330031-2.20556943
```

```
Out[3]: -9.733466996930673e-05
```

A take home message and a word of caution:

- The exact error, $|p_n - p|$, is guaranteed to be less than 10^{-4} after 23 iterations from Corollary 42, but as we observed in this example, this could happen before 23 iterations.
- The stopping criterion used in the code is based on $|p_n - p_{n-1}|$, not $|p_n - p|$, so the iteration number that makes these quantities less than a tolerance ϵ will not be the same in general.

Theorem 44. *Assume p is a solution of $g(x) = x$, and suppose $g(x)$ is continuously differentiable in some interval about p with $|g'(p)| < 1$. Then the fixed-point iteration converges to p, provided p_0 is chosen sufficiently close to p. Moreover, the convergence is linear if $g'(p) \neq 0$.*

Proof. Since g' is continuous and $|g'(p)| < 1$, there exists an interval $I = [p-\epsilon, p+\epsilon]$ such that $|g'(x)| \leq k$ for all $x \in I$, for some $k < 1$. Then, from Remark 39, we know $|g(x) - g(y)| \leq k|x-y|$ for all $x, y \in I$. Next, we argue that $g(x) \in I$ if $x \in I$. Indeed, if $|x-p| < \epsilon$, then

$$|g(x) - p| = |g(x) - g(p)| \leq |g'(\xi)||x-p| < k\epsilon < \epsilon$$

hence $g(x) \in I$. Now use Theorem 40, setting $[a, b]$ to $[p-\epsilon, p+\epsilon]$, to conclude the fixed-point iteration converges.

To prove convergence is linear, we note

$$|p_{n+1} - p| = |g(p_n) - g(p)| \leq |g'(\xi_n)||p_n - p| \leq k|p_n - p|$$

which is the definition of linear convergence (with k being a positive constant less than 1).

We can actually prove something more:

$$\lim_{n\to\infty} \frac{|p_{n+1}-p|}{|p_n-p|} = \lim_{n\to\infty} \frac{|g(p_n)-g(p)|}{|p_n-p|} = \lim_{n\to\infty} \frac{|g'(\xi_n)||p_n-p|}{|p_n-p|} = \lim_{n\to\infty} |g'(\xi_n)| = |g'(p)|.$$

The last equality follows since g' is continuous, and $\xi_n \to p$, which is a consequence of ξ_n being between p and p_n, and $p_n \to p$, as $n \to \infty$. □

Example 45. Let $g(x) = x + c(x^2 - 2)$, which has the fixed-point $p = \sqrt{2} \approx 1.4142$. Pick a value for c to ensure the convergence of fixed-point iteration. For the picked value c, determine the interval of convergence $I = [a, b]$, that is, the interval for which any p_0 from the interval gives rise to a converging fixed-point iteration. Then write a Python code to test the results.

Solution. Theorem 44 requires $|g'(p)| < 1$. We have $g'(x) = 1 + 2xc$, and thus $g'(\sqrt{2}) = 1 + 2\sqrt{2}c$. Therefore

$$\begin{aligned} |g'(\sqrt{2})| < 1 &\Rightarrow -1 < 1 + 2\sqrt{2}c < 1 \\ &\Rightarrow -2 < 2\sqrt{2}c < 0 \\ &\Rightarrow \frac{-1}{\sqrt{2}} < c < 0. \end{aligned}$$

Any c from this interval works: let's pick $c = -1/4$.

Now we need to find an interval $I = [\sqrt{2} - \epsilon, \sqrt{2} + \epsilon]$ such that

$$|g'(x)| = |1 + 2xc| = \left|1 - \frac{x}{2}\right| \leq k$$

for some $k < 1$, for all $x \in I$. Plot $g'(x)$ and observe that one choice is $\epsilon = 0.1$, so that $I = [\sqrt{2}-0.1, \sqrt{2}+0.1] = [1.3142, 1.5142]$. Since $g'(x)$ is positive and decreasing on $I = [1.3142, 1.5142]$, $|g'(x)| \leq 1 - \frac{1.3142}{2} = 0.3429 < 1$, for any $x \in I$. Then any starting value x_0 from I gives convergence.

For $c = -1/4$, the function becomes $g(x) = x - \frac{x^2-2}{4}$. Pick $p_0 = 1.5$ as the starting point. Using the fixed-point iteration code of the previous example, we obtain:

```
In [4]: fixedpt(lambda x: x-(x**2-2)/4, 1.5, 1e-5, 15)

p is  1.414214788550556  and the iteration number is  9
```

The absolute error is:

```
In [5]: 1.414214788550556-(2**.5)

Out[5]: 1.2261774609001463e-06
```

Let's experiment with other starting values. Although $p_0 = 2$ is not in the interval of convergence I, we expect convergence since $g'(2) = 0$:

```
In [6]: fixedpt(lambda x: x-(x**2-2)/4, 2, 1e-5, 15)

p is  1.414214788550556  and the iteration number is  10
```

Let's try $p_0 = -5$. Note that this is not only outside the interval of convergence I, but $g'(-5) = 3.5 > 1$, so we do not expect convergence.

```
In [7]: fixedpt(lambda x: x-(x**2-2)/4, -5, 1e-5, 15)


        ---------------------------------------------------------------------------

        OverflowError                                Traceback (most recent call last)

        <ipython-input-8-2df659a6b4d1> in <module>
        ----> 1 fixedpt(lambda x: x-(x**2-2)/4, -5, 1e-5, 15)


        <ipython-input-2-d29380b04f2a> in fixedpt(g, pzero, eps, N)
          2     n = 1
          3     while n<N:
        ----> 4         pone = g(pzero)
          5         if np.abs(pone-pzero)<eps:
          6             print('p is ', pone, ' and the iteration number is ', n)
```

```
        <ipython-input-8-2df659a6b4d1> in <lambda>(x)
    ----> 1 fixedpt(lambda x: x-(x**2-2)/4, -5, 1e-5, 15)


        OverflowError: (34, 'Result too large')
```

Let's verify the linear convergence of the fixed-point iteration numerically in this example. We write another version of the fixed-point code, **fixedpt2**, and we compute $\frac{p_n-\sqrt{2}}{p_{n-1}-\sqrt{2}}$ for each n.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [2]: def fixedpt2(g, pzero, eps, N):
            n = 1
            arr = np.array([])
            error = 1.
            while n<N and error>1e-5:
                pone = g(pzero)
                error = np.abs(pone-pzero)
                arr = np.append(arr, (pone-2**0.5)/(pzero-2**0.5))
                pzero = pone
                n += 1
            return arr
```

```
In [3]: arr = fixedpt2(lambda x: x-(x**2-2)/4, 1.5, 1e-7, 15)
        arr
```

```
Out[3]: array([0.27144661, 0.28707161, 0.291222  , 0.29240652, 0.29275091,
               0.29285156, 0.29288102, 0.29288965, 0.29289217])
```

```
In [4]: plt.plot(arr);
```

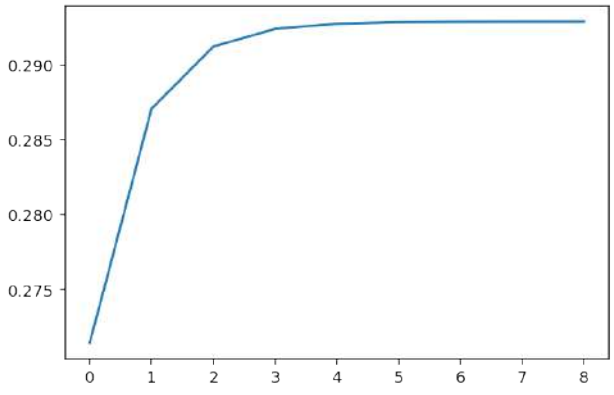


Figure 2.6: Fixed-point iteration

The graph suggests the limit of $\frac{p_n-\sqrt{2}}{p_{n-1}-\sqrt{2}}$ exists and it is around 0.295, supporting linear convergence.

2.7 High-order fixed-point iteration

In the proof of Theorem 44, we showed

$$\lim_{n\to\infty} \frac{|p_{n+1}-p|}{|p_n-p|} = |g'(p)|$$

which implied that the fixed-point iteration has linear convergence, if $g'(p) \neq 0$.

If this limit were zero, then we would have

$$\lim_{n\to\infty} \frac{|p_{n+1}-p|}{|p_n-p|} = 0,$$

which means the denominator is growing at a larger rate than the numerator. We could then ask if

$$\lim_{n\to\infty} \frac{|p_{n+1}-p|}{|p_n-p|^\alpha} = \text{nonzero constant}$$

for some $\alpha > 1$.

Theorem 46. *Assume p is a solution of $g(x) = x$ where $g \in C^\alpha(I)$ for some interval I that contains p, and for some $\alpha \geq 2$. Furthermore assume*

$$g'(p) = g''(p) = ... = g^{(\alpha-1)}(p) = 0, \text{ and } g^{(\alpha)}(p) \neq 0.$$

Then if the initial guess p_0 is sufficiently close to p, the fixed-point iteration $p_n = g(p_{n-1}), n \geq 1$, will have order of convergence of α, and

$$\lim_{n\to\infty} \frac{p_{n+1} - p}{(p_n - p)^\alpha} = \frac{g^{(\alpha)}(p)}{\alpha!}.$$

Proof. From Taylor's theorem,

$$p_{n+1} = g(p_n) = g(p) + (p_n - p)g'(p) + ... + \frac{(p_n - p)^{\alpha-1}}{(\alpha-1)!} g^{(\alpha-1)}(p) + \frac{(p_n - p)^\alpha}{\alpha!} g^{(\alpha)}(\xi_n)$$

where ξ_n is a number between p_n and p, and all numbers are in I. From the hypothesis, this simplifies as

$$p_{n+1} = p + \frac{(p_n - p)^\alpha}{\alpha!} g^{(\alpha)}(\xi_n) \Rightarrow \frac{p_{n+1} - p}{(p_n - p)^\alpha} = \frac{g^{(\alpha)}(\xi_n)}{\alpha!}.$$

From Theorem 44, if p_0 is chosen sufficiently close to p, then $\lim_{n\to\infty} p_n = p$. The order of convergence is α with

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|^\alpha} = \lim_{n\to\infty} \frac{|g^{(\alpha)}(\xi_n)|}{\alpha!} = \frac{|g^{(\alpha)}(p)|}{\alpha!} \neq 0.$$

□

Application to Newton's Method

Recall Newton's iteration

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}.$$

Put $g(x) = x - \frac{f(x)}{f'(x)}$. Then the fixed-point iteration $p_n = g(p_{n-1})$ is Newton's method. We have

$$g'(x) = 1 - \frac{[f'(x)]^2 - f(x)f''(x)}{[f'(x)]^2} = \frac{f(x)f''(x)}{[f'(x)]^2}$$

and thus

$$g'(p) = \frac{f(p)f''(p)}{[f'(p)]^2} = 0.$$

Similarly,

$$g''(x) = \frac{(f'(x)f''(x) + f(x)f'''(x))\,(f'(x))^2 - f(x)f''(x)2f'(x)f''(x)}{[f'(x)]^4}$$

which implies

$$g''(p) = \frac{(f'(p)f''(p))\,(f'(p))^2}{[f'(p)]^4} = \frac{f''(p)}{f'(p)}.$$

If $f''(p) \neq 0$, then Theorem 46 implies Newton's method has quadratic convergence with

$$\lim_{n\to\infty} \frac{p_{n+1} - p}{(p_n - p)^2} = \frac{f''(p)}{2f'(p)}$$

which was proved earlier in Theorem 32.

Exercise 2.7-1: Use Theorem 38 (and Remark 39) to show that $g(x) = 3^{-x}$ has a unique fixed-point on $[1/4, 1]$. Use Corollary 42, part (4), to find the number of iterations necessary to achieve 10^{-5} accuracy. Then use the Python code to obtain an approximation, and compare the error with the theoretical estimate obtained from Corollary 42.

Exercise 2.7-2: Let $g(x) = 2x - cx^2$ where c is a positive constant. Prove that if the fixed-point iteration $p_n = g(p_{n-1})$ converges to a non-zero limit, then the limit is $1/c$.

Chapter 3

Interpolation

In this chapter, we will study the following problem: given data $(x_i, y_i), i = 0, 1, ..., n$, find a function f such that $f(x_i) = y_i$. This problem is called the interpolation problem, and f is called the interpolating function, or interpolant, for the given data.

Interpolation is used, for example, when we use mathematical software to plot a smooth curve through discrete data points, when we want to find the in-between values in a table, or when we differentiate or integrate black-box type functions.

How do we choose f? Or, what kind of function do we want f to be? There are several options. Examples of functions used in interpolation are polynomials, piecewise polynomials, rational functions, trigonometric functions, and exponential functions. As we try to find a good choice for f for our data, some questions to consider are whether we want f to inherit the properties of the data (for example, if the data is periodic, should we use a trigonometric function as f?), and how we want f behave between data points. In general f should be easy to evaluate, and easy to integrate & differentiate.

Here is a general framework for the interpolation problem. We are given data, and we pick a family of functions from which the interpolant f will be chosen:

- Data: $(x_i, y_i), i = 0, 1, ..., n$
- Family: Polynomials, trigonometric functions, etc.

Suppose the family of functions selected forms a vector space. Pick a basis for the vector space: $\phi_0(x), \phi_1(x), ..., \phi_n(x)$. Then the interpolating function can be written as a linear combination of the basis vectors (functions):

$$f(x) = \sum_{k=0}^{n} a_k \phi_k(x).$$

We want f to pass through the data points, that is, $f(x_i) = y_i$. Then determine a_k so that:

$$f(x_i) = \sum_{k=0}^{n} a_k \phi_k(x_i) = y_i, i = 0, 1, ..., n,$$

which is a system of $n+1$ equations with $n+1$ unknowns. Using matrices, the problem is to solve the matrix equation

$$Aa = y$$

for a, where

$$A = \begin{bmatrix} \phi_0(x_0) & ... & \phi_n(x_0) \\ \phi_0(x_1) & ... & \phi_n(x_1) \\ \vdots & & \\ \phi_0(x_n) & ... & \phi_n(x_n) \end{bmatrix}, a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

3.1 Polynomial interpolation

In polynomial interpolation, we pick polynomials as the family of functions in the interpolation problem.

- Data: $(x_i, y_i), i = 0, 1, ..., n$
- Family: Polynomials

The space of polynomials up to degree n is a vector space. We will consider three choices for the basis for this vector space:

- Basis:
  - Monomial basis: $\phi_k(x) = x^k$
  - Lagrange basis: $\phi_k(x) = \prod_{j=0, j\neq k}^{n} \left(\frac{x - x_j}{x_k - x_j}\right)$
  - Newton basis: $\phi_k(x) = \prod_{j=0}^{k-1}(x - x_j)$

  where $k = 0, 1, ..., n$.

Once we decide on the basis, the interpolating polynomial can be written as a linear combination of the basis functions:

$$p_n(x) = \sum_{k=0}^{n} a_k \phi_k(x)$$

where $p_n(x_i) = y_i, i = 0, 1, ..., n$.

Here is an important question. How do we know that p_n, a polynomial of degree at most n passing through the data points, actually exists? Or, equivalently, how do we know the system of equations $p_n(x_i) = y_i, i = 0, 1, ..., n$, has a solution?

The answer is given by the following theorem, which we will prove later in this section.

Theorem 47. *If points $x_0, x_1, ..., x_n$ are distinct, then for real values $y_0, y_1, ..., y_n$, there is a unique polynomial p_n of degree at most n such that $p_n(x_i) = y_i, i = 0, 1, ..., n$.*

We mentioned three families of basis functions for polynomials. The choice of a family of basis functions affects:

- The accuracy of the numerical methods to solve the system of linear equations $Aa = y$.
- The ease at which the resulting polynomial can be evaluated, differentiated, integrated, etc.

Monomial form of polynomial interpolation

Given data $(x_i, y_i), i = 0, 1, ..., n$, we know from the previous theorem that there exists a polynomial $p_n(x)$ of degree at most n, that passes through the data points. To represent $p_n(x)$, we will use the monomial basis functions, $1, x, x^2, ..., x^n$, or written more succinctly,

$$\phi_k(x) = x^k, k = 0, 1, ..., n.$$

The interpolating polynomial $p_n(x)$ can be written as a linear combination of these basis functions as

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n.$$

We will determine a_i using the fact that p_n is an interpolant for the data:

$$p_n(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + ... + a_n x_i^n = y_i$$

for $i = 0, 1, ..., n$. Or, in matrix form, we want to solve

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & ... & x_0^n \\ 1 & x_1 & x_1^2 & & x_1^n \\ \vdots & & & & \\ 1 & x_n & x_n^2 & & x_n^n \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}}_{y}$$

for $[a_0, ..., a_n]^T$ where $[\cdot]^T$ stands for the transpose of the vector. The coefficient matrix A is known as the van der Monde matrix. This is usually an ill-conditioned matrix, which means solving the system of equations could result in large error in the coefficients a_i. An intuitive way to understand the ill-conditioning is to plot several basis monomials, and note how less distinguishable they are as the degree increases, making the columns of the matrix nearly linearly dependent.

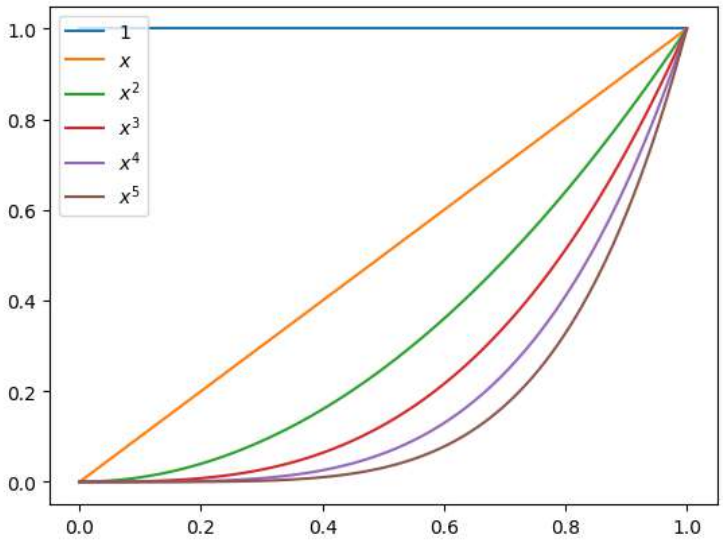


Figure 3.1: Monomial basis functions

Solving the matrix equation $Aa = b$ could also be expensive. Using Gaussian elimination to solve the matrix equation for a general matrix A requires $O(n^3)$ operations. This means the number of operations grows like Cn^3, where C is a positive constant.[1] However, there are some advantages to the monomial form: evaluating the polynomial is very efficient using Horner's method, which is the nested form discussed in Exercises 1.3-4, 1.3-5 of Chapter 1, requiring $O(n)$ operations. Differentiation and integration are also relatively efficient.

Lagrange form of polynomial interpolation

The ill-conditioning of the van der Monde matrix, as well as the high complexity of solving the resulting matrix equation in the monomial form of polynomial interpolation, motivate us to explore other basis functions for polynomials. As before, we start with data $(x_i, y_i), i = 0, 1, ..., n$, and call our interpolating polynomial of degree at most n, $p_n(x)$. The Lagrange basis functions up to degree n (also called cardinal polynomials) are defined as

$$l_k(x) = \prod_{j=0, j\neq k}^{n} \left(\frac{x - x_j}{x_k - x_j} \right), k = 0, 1, ..., n.$$

We write the interpolating polynomial $p_n(x)$ as a linear combination of these basis functions as

$$p_n(x) = a_0 l_0(x) + a_1 l_1(x) + ... + a_n l_n(x).$$

[1] The formal definition of the big O notation is as follows: We write $f(n) = O(g(n))$ as $n \to \infty$ if and only if there exists a positive constant M and a positive integer n^* such that $|f(n)| \leq Mg(n)$ for all $n \geq n^*$.

We will determine a_i from

$$p_n(x_i) = a_0 l_0(x_i) + a_1 l_1(x_i) + ... + a_n l_n(x_i) = y_i$$

for $i = 0, 1, ..., n$. Or, in matrix form, we want to solve

$$\underbrace{\begin{bmatrix} l_0(x_0) & l_1(x_0) & ... & l_n(x_0) \\ l_0(x_1) & l_1(x_1) & & l_n(x_1) \\ \vdots & & & \\ l_0(x_n) & l_1(x_n) & & l_n(x_n) \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}}_{y}$$

for $[a_0, ..., a_n]^T$.

Solving this matrix equation is trivial for the following reason. Observe that $l_k(x_k) = 1$ and $l_k(x_i) = 0$ for all $i \neq k$. Then the coefficient matrix A becomes the identity matrix, and

$$a_k = y_k \text{ for } k = 0, 1, ..., n.$$

The interpolating polynomial becomes

$$p_n(x) = y_0 l_0(x) + y_1 l_1(x) + ... + y_n l_n(x).$$

The main advantage of the Lagrange form of interpolation is that finding the interpolating polynomial is trivial: there is no need to solve a matrix equation. However, the evaluation, differentiation, and integration of the Lagrange form of a polynomial is more expensive than, for example, the monomial form.

Example 48. Find the interpolating polynomial using the monomial basis and Lagrange basis functions for the data: $(-1, -6), (1, 0), (2, 6)$.

- Monomial basis: $p_2(x) = a_0 + a_1 x + a_2 x^2$

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}}_{y} \Rightarrow \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} -6 \\ 0 \\ 6 \end{bmatrix}$$

We can use Gaussian elimination to solve this matrix equation, or get help from Python:

```
In [1]: import numpy as np

In [2]: A = np.array([[1,-1,1], [1,1,1], [1,2,4]])
        A
```

```
Out[2]: array([[ 1, -1,  1],
               [ 1,  1,  1],
               [ 1,  2,  4]])

In [3]: y = np.array([-6, 0, 6])
        y

Out[3]: array([-6,  0,  6])

In [4]: np.linalg.solve(A, y)

Out[4]: array([-4.,  3.,  1.])
```

Since the solution is $a = [-4, 3, 1]^T$, we obtain

$$p_2(x) = -4 + 3x + x^2.$$

- Lagrange basis: $p_2(x) = y_0 l_0(x) + y_1 l_1(x) + y_2 l_2(x) = -6l_0(x) + 0l_1(x) + 6l_2(x)$ where

$$l_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{(x - 1)(x - 2)}{(-1 - 1)(-1 - 2)} = \frac{(x - 1)(x - 2)}{6}$$
$$l_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{(x + 1)(x - 1)}{(2 + 1)(2 - 1)} = \frac{(x + 1)(x - 1)}{3}$$

therefore

$$p_2(x) = -6\frac{(x - 1)(x - 2)}{6} + 6\frac{(x + 1)(x - 1)}{3} = -(x - 1)(x - 2) + 2(x + 1)(x - 1).$$

If we multiply out and collect the like terms, we obtain $p_2(x) = -4 + 3x + x^2$, which is the polynomial we obtained from the monomial basis earlier.

Exercise 3.1-1: Prove that $\sum_{k=0}^{n} l_k(x) = 1$ for all x, where l_k are the Lagrange basis functions for $n + 1$ data points. (Hint. First verify the identity for $n = 1$ algebraically, for any two data points. For the general case, think about what special function's interpolating polynomial in Lagrange form is $\sum_{k=0}^{n} l_k(x)$).

Newton's form of polynomial interpolation

The Newton basis functions up to degree n are

$$\pi_k(x) = \prod_{j=0}^{k-1}(x - x_j), k = 0, 1, ..., n$$

where $\pi_0(x) = \prod_{j=0}^{-1}(x - x_j)$ is interpreted as 1. The interpolating polynomial p_n, written as a linear combination of Newton basis functions, is

$$\begin{aligned} p_n(x) &= a_0\pi_0(x) + a_1\pi_1(x) + ... + a_n\pi_n(x) \\ &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + ... + a_n(x - x_0)\cdots(x - x_{n-1}). \end{aligned}$$

We will determine a_i from

$$p_n(x_i) = a_0 + a_1(x_i - x_0) + ... + a_n(x_i - x_0)\cdots(x_i - x_{n-1}) = y_i,$$

for $i = 0, 1, ..., n$, or in matrix form

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & (x_1 - x_0) & 0 & & 0 \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & (x_n - x_0) & (x_n - x_0)(x_n - x_1) & \dots & \prod_{i=0}^{n-1}(x_n - x_i) \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}}_{y}$$

for $[a_0, ..., a_n]^T$. Note that the coefficient matrix A is lower-triangular, and a can be solved by forward substitution, which is shown in the next example, in $O(n^2)$ operations.

Example 49. Find the interpolating polynomial using Newton's basis for the data: $(-1, -6), (1, 0), (2, 6)$.

Solution. We have $p_2(x) = a_0 + a_1\pi_1(x) + a_2\pi_2(x) = a_0 + a_1(x + 1) + a_2(x + 1)(x - 1)$. Find a_0, a_1, a_2 from

$$\begin{aligned} p_2(-1) = -6 \Rightarrow a_0 + a_1(-1 + 1) + a_2(-1 + 1)(-1 - 1) = a_0 = -6 \\ p_2(1) = 0 \Rightarrow a_0 + a_1(1 + 1) + a_2(1 + 1)(1 - 1) = a_0 + 2a_1 = 0 \\ p_2(2) = 6 \Rightarrow a_0 + a_1(2 + 1) + a_2(2 + 1)(2 - 1) = a_0 + 3a_1 + 3a_2 = 6 \end{aligned}$$

or, in matrix form

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 3 & 3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} -6 \\ 0 \\ 6 \end{bmatrix}.$$

Forward substitution is:

$$a_0 = -6$$
$$a_0 + 2a_1 = 0 \Rightarrow -6 + 2a_1 = 0 \Rightarrow a_1 = 3$$
$$a_0 + 3a_1 + 3a_2 = 6 \Rightarrow -6 + 9 + 3a_2 = 6 \Rightarrow a_2 = 1.$$

Therefore $a = [-6, 3, 1]^T$ and

$$p_2(x) = -6 + 3(x+1) + (x+1)(x-1).$$

Factoring out and simplifying gives $p_2(x) = -4 + 3x + x^2$, which is the polynomial discussed in Example 48.

Summary: The interpolating polynomial $p_2(x)$ for the data, $(-1, -6), (1, 0), (2, 6)$, represented in three different basis functions is:

$$\text{Monomial: } p_2(x) = -4 + 3x + x^2$$
$$\text{Lagrange: } p_2(x) = -(x-1)(x-2) + 2(x+1)(x-1)$$
$$\text{Newton: } p_2(x) = -6 + 3(x+1) + (x+1)(x-1)$$

Similar to the monomial form, a polynomial written in Newton's form can be evaluated using the Horner's method which has $O(n)$ complexity:

$$\begin{aligned} p_n(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}) \\ &= a_0 + (x - x_0)(a_1 + (x - x_1)(a_2 + \ldots + (x - x_{n-2})(a_{n-1} + (x - x_{n-1})(a_n))\cdots)) \end{aligned}$$

Example 50. Write $p_2(x) = -6 + 3(x+1) + (x+1)(x-1)$ using the nested form.

Solution. $-6 + 3(x+1) + (x+1)(x-1) = -6 + (x+1)(2+x)$; note that the left-hand side has 2 multiplications, and the right-hand side has 1.

Complexity of the three forms of polynomial interpolation: The number of multiplications required in solving the corresponding matrix equation in each polynomial basis is:

- Monomial $\rightarrow O(n^3)$
- Lagrange $\rightarrow$ trivial

- Newton $\rightarrow O(n^2)$

Evaluating the polynomials can be done efficiently using Horner's method for monomial and Newton forms. A modified version of Lagrange form can also be evaluated using Horner's method, but we do not discuss it here.

Exercise 3.1-2: Compute, by hand, the interpolating polynomial to the data $(-1, 0)$, $(0.5, 1)$, $(1, 0)$ using the monomial, Lagrange, and Newton basis functions. Verify the three polynomials are identical.

It's time to discuss some theoretical results for polynomial interpolation. Let's start with proving Theorem 47 which we stated earlier:

Theorem. *If points $x_0, x_1, ..., x_n$ are distinct, then for real values $y_0, y_1, ..., y_n$, there is a unique polynomial p_n of degree at most n such that $p_n(x_i) = y_i, i = 0, 1, ..., n$.*

Proof. We have already established the existence of p_n without mentioning it! The Lagrange form of the interpolating polynomial constructs p_n directly:

$$p_n(x) = \sum_{k=0}^{n} y_k l_k(x) = \sum_{k=0}^{n} y_k \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_k - x_j}.$$

Let's prove uniqueness. Assume p_n, q_n are two distinct polynomials satisfying the conclusion. Then $p_n - q_n$ is a polynomial of degree at most n such that $(p_n - q_n)(x_i) = 0$ for $i = 0, 1, ..., n$. This means the non-zero polynomial $(p_n - q_n)$ of degree at most n, has $(n + 1)$ distinct roots, which is a contradiction. □

The following theorem, which we state without proof, establishes the error of polynomial interpolation. Notice the similarities between this and Taylor's Theorem 7.

Theorem 51. *Let $x_0, x_1, ..., x_n$ be distinct numbers in the interval $[a, b]$ and $f \in C^{n+1}[a, b]$. Then for each $x \in [a, b]$, there is a number ξ between $x_0, x_1, ..., x_n$ such that*

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n + 1)!}(x - x_0)(x - x_1) \cdots (x - x_n).$$

The following lemma is useful in finding upper bounds for $|f(x) - p_n(x)|$ using Theorem 51, when the nodes $x_0, ..., x_n$ are **equally spaced**.

Lemma 52. *Consider the partition of* $[a,b]$ *as* $x_0 = a, x_1 = a+h, ..., x_n = a+nh = b$. *More succinctly,* $x_i = a+ih$ *for* $i = 0, 1, ..., n$ *and* $h = \frac{b-a}{n}$. *Then for any* $x \in [a,b]$

$$\prod_{i=0}^{n} |x - x_i| \leq \frac{1}{4} h^{n+1} n!$$

Proof. Since $x \in [a,b]$, it falls into one of the subintervals: let $x \in [x_j, x_{j+1}]$. Consider the product $|x-x_j||x-x_{j+1}|$. Put $s = |x-x_j|$ and $t = |x-x_{j+1}|$. The maximum of st given $s+t=h$, using Calculus, can be found to be $h^2/4$, which is attained when x is the midpoint, and thus $s = t = h/2$. Then

$$\begin{aligned}
\prod_{i=0}^{n} |x - x_i| &= |x-x_0| \cdots |x-x_{j-1}||x-x_j||x-x_{j+1}||x-x_{j+2}| \cdots |x-x_n| \\
&\leq |x-x_0| \cdots |x-x_{j-1}| \frac{h^2}{4} |x-x_{j+2}| \cdots |x-x_n| \\
&\leq |x_{j+1}-x_0| \cdots |x_{j+1}-x_{j-1}| \frac{h^2}{4} |x_j - x_{j+2}| \cdots |x_j - x_n| \\
&\leq (j+1)h \cdots 2h \left(\frac{h^2}{4}\right) (2h) \cdots (n-j)h \\
&= h^j (j+1)! \frac{h^2}{4} (n-j)! h^{n-j-1} \\
&\leq h^{n+1} \frac{n!}{4}.
\end{aligned}$$

□

Example 53. Find an upper bound for the absolute error when $f(x) = \cos x$ is approximated by its interpolating polynomial $p_n(x)$ on $[0, \pi/2]$. For the interpolating polynomial, use 5 equally spaced nodes ($n = 4$) in $[0, \pi/2]$, including the endpoints.

Solution. From Theorem 51,

$$|f(x) - p_4(x)| = \frac{|f^{(5)}(\xi)|}{5!} |(x-x_0) \cdots (x-x_4)|.$$

We have $|f^{(5)}(\xi)| \leq 1$. The nodes are equally spaced with $h = (\pi/2 - 0)/4 = \pi/8$. Then from the previous lemma,

$$|(x-x_0) \cdots (x-x_4)| \leq \frac{1}{4} \left(\frac{\pi}{8}\right)^5 4!$$

and therefore

$$|f(x) - p_4(x)| \leq \frac{1}{5!} \frac{1}{4} \left(\frac{\pi}{8}\right)^5 4! = 4.7 \times 10^{-4}.$$

Exercise 3.1-3: Find an upper bound for the absolute error when $f(x) = \ln x$ is approximated by an interpolating polynomial of degree five with six nodes equally spaced in the interval $[1, 2]$.

We now revisit Newton's form of interpolation, and learn an alternative method, known as **divided differences**, to compute the coefficients of the interpolating polynomial. This approach is numerically more stable than the forward substitution approach we used earlier. Let's recall the interpolation problem.

- Data: $(x_i, y_i), i = 0, 1, ..., n$
- Interpolant in Newton's form:

$$p_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + ... + a_n(x - x_0) \cdots (x - x_{n-1})$$

  Determine a_i from $p_n(x_i) = y_i, i = 0, 1, ..., n$.

Let's think of the y-coordinates of the data, y_i, as values of an unknown function f evaluated at x_i, i.e., $f(x_i) = y_i$. Substitute $x = x_0$ in the interpolant to get:

$$a_0 = f(x_0).$$

Substitute $x = x_1$ to get $a_0 + a_1(x_1 - x_0) = f(x_1)$ or,

$$a_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}.$$

Substitute $x = x_2$ to get, after some algebra

$$a_2 = \frac{\frac{f(x_2)-f(x_0)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}\frac{x_2-x_0}{x_2-x_1}}{x_2 - x_0}$$

which can be further rewritten as

$$a_2 = \frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2 - x_0}.$$

Inspecting the formulas for a_0, a_1, a_2 suggests the following simplified new notation called **divided differences**:

$$a_0 = f(x_0) = f[x_0] \longrightarrow \text{0th divided difference}$$

$$a_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = f[x_0, x_1] \longrightarrow \text{1st divided difference}$$

$$a_2 = \frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2 - x_0} = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = f[x_0, x_1, x_2] \longrightarrow \text{2nd divided difference}$$

And in general, a_k will be given by the kth divided difference:

$$a_k = f[x_0, x_1, ..., x_k].$$

With this new notation, Newton's interpolating polynomial can be written as

$$\begin{aligned} p_n(x) = f[x_0] + \sum_{k=1}^{n} f[x_0, x_1, ..., x_k](x - x_0) \cdots (x - x_{k-1}) \\ = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \dots \\ + f[x_0, x_1, ..., x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1}) \end{aligned}$$

Here is the formal definition of divided differences:

Definition 54. Given data $(x_i, f(x_i)), i = 0, 1, ..., n$, the divided differences are defined recursively as

$$\begin{aligned} f[x_0] &= f(x_0) \\ f[x_0, x_1, ..., x_k] &= \frac{f[x_1, ..., x_k] - f[x_0, ..., x_{k-1}]}{x_k - x_0} \end{aligned}$$

where $k = 0, 1, ..., n$.

Theorem 55. *The ordering of the data in constructing divided differences is not important, that is, the divided difference $f[x_0, ..., x_k]$ is invariant under all permutations of the arguments $x_0, ..., x_k$.*

Proof. Consider the data $(x_0, y_0), (x_1, y_1), ..., (x_k, y_k)$ and let $p_k(x)$ be its interpolating polynomial:

$$\begin{aligned} p_k(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + ... \\ + f[x_0, ..., x_k](x - x_0) \cdots (x - x_{k-1}). \end{aligned}$$

Now let's consider a permutation of the x_i; let's label them as $\tilde{x}_0, \tilde{x}_1, ..., \tilde{x}_k$. The interpolating polynomial for the permuted data does not change, since the data $x_0, x_1, ..., x_k$ (omitting the y-coordinates) is the same as $\tilde{x}_0, \tilde{x}_1, ..., \tilde{x}_k$, just in different order. Therefore

$$\begin{aligned} p_k(x) = f[\tilde{x}_0] + f[\tilde{x}_0, \tilde{x}_1](x - \tilde{x}_0) + f[\tilde{x}_0, \tilde{x}_1, \tilde{x}_2](x - \tilde{x}_0)(x - \tilde{x}_1) + ... \\ + f[\tilde{x}_0, ..., \tilde{x}_k](x - \tilde{x}_0) \cdots (x - \tilde{x}_{k-1}). \end{aligned}$$

The coefficient of the polynomial $p_k(x)$ for the highest degree x^k is $f[x_0, ..., x_k]$ in the first equation, and $f[\tilde{x}_0, ..., \tilde{x}_k]$ in the second. Therefore they must be equal to each other. □

Example 56. Find the interpolating polynomial for the data $(-1, -6), (1, 0), (2, 6)$ using Newton's form and divided differences.

Solution. We want to compute

$$p_2(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1).$$

Here are the finite differences:

x	$f[x]$	First divided difference	Second divided difference
$x_0 = -1$	$f[x_0] = -6$		
		$f[x_0, x_1] = \frac{f[x_1]-f[x_0]}{x_1-x_0} = 3$	
$x_1 = 1$	$f[x_1] = 0$		$f[x_0, x_1, x_2] = \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} = 1$
		$f[x_1, x_2] = \frac{f[x_2]-f[x_1]}{x_2-x_1} = 6$	
$x_2 = 2$	$f[x_2] = 6$		

Therefore

$$p_2(x) = -6 + 3(x + 1) + 1(x + 1)(x - 1),$$

which is the same polynomial we had in Example 49.

Exercise 3.1-4: Consider the function f given in the following table.

x	1	2	4	6
$f(x)$	2	3	5	9

a) Construct a divided difference table for f by hand, and write the Newton form of the interpolating polynomial using the divided differences.

b) Assume you are given a new data point for the function: $x = 3, y = 4$. Find the new interpolating polynomial. (Hint: Think about how to update the interpolating polynomial you found in part (a).)

c) If you were working with the Lagrange form of the interpolating polynomial instead of the Newton form, and you were given an additional data point like in part (b), how easy would it be (compared to what you did in part (b)) to update your interpolating polynomial?

Example 57. Before the widespread availability of computers and mathematical software, the values of some often-used mathematical functions were disseminated to researchers and engineers via tables. The following table, taken from [1], displays some values of the gamma function, $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$.

x	1.750	1.755	1.760	1.765
$\Gamma(x)$	0.91906	0.92021	0.92137	0.92256

Use polynomial interpolation to estimate $\Gamma(1.761)$.

Solution. The finite differences, with five-digit rounding, are:

i	x_i	$f[x_i]$	$f[x_i, x_{i+1}]$	$f[x_{i-1}, x_i, x_{i+1}]$	$f[x_0, x_1, x_2, x_3]$
0	1.750	0.91906			
			0.23		
1	1.755	0.92021		0.2	
			0.232		26.667
2	1.760	0.92137		0.6	
			0.238		
3	1.765	0.92256			

Here are various estimates for $\Gamma(1.761)$ using interpolating polynomials of increasing degrees:

$$p_1(x) = f[x_0] + f[x_0, x_1](x - x_0) \Rightarrow p_1(1.761) = 0.91906 + 0.23(1.761 - 1.750) = 0.92159$$

$$p_2(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1)$$

$$\Rightarrow p_2(1.761) = 0.92159 + (0.2)(1.761 - 1.750)(1.761 - 1.755) = 0.9216$$

$$p_3(x) = p_2(x) + f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2)$$

$$\Rightarrow p_3(1.761) = 0.9216 + 26.667(1.761 - 1.750)(1.761 - 1.755)(1.761 - 1.760) = 0.9216$$

Next we will change the ordering of the data and repeat the calculations. We will list the data in decreasing order of the x-coordinates:

i	x_i	$f[x_i]$	$f[x_i, x_{i+1}]$	$f[x_{i-1}, x_i, x_{i+1}]$	$f[x_0, x_1, x_2, x_3]$
0	1.765	0.92256			
			0.238		
1	1.760	0.92137		0.6	
			0.232		26.667
2	1.755	0.92021		0.2	
			0.23		
3	1.750	0.91906			

The polynomial evaluations are:

$$p_1(1.761) = 0.92256 + 0.238(1.761 - 1.765) = 0.92161$$
$$p_2(1.761) = 0.92161 + 0.6(1.761 - 1.765)(1.761 - 1.760) = 0.92161$$
$$p_3(1.761) = 0.92161 + 26.667(1.761 - 1.765)(1.761 - 1.760)(1.761 - 1.755) = 0.92161$$

Summary of results: The following table displays the results for each ordering of the data, together with the correct $\Gamma(1.761)$ to 7 digits of accuracy.

Ordering	$(1.75, 1.755, 1.76, 1.765)$	$(1.765, 1.76, 1.755, 1.75)$
$p_1(1.761)$	0.92159	0.92161
$p_2(1.761)$	0.92160	0.92161
$p_3(1.761)$	0.92160	0.92161
$\Gamma(1.761)$	0.9216103	0.9216103

Exercise 3.1-5: Answer the following questions:

a) Theorem 55 stated that the ordering of the data in divided differences does not matter. But we see differences in the two tables above. Is this a contradiction?

b) $p_1(1.761)$ is a better approximation to $\Gamma(1.761)$ in the second ordering. Is this expected?

c) $p_3(1.761)$ is different in the two orderings, however, this difference is due to rounding error. In other words, if the calculations can be done exactly, $p_3(1.761)$ will be the same in each ordering of the data. Why?

Exercise 3.1-6: Consider a function $f(x)$ such that $f(2) = 1.5713$, $f(3) = 1.5719, f(5) = 1.5738$, and $f(6) = 1.5751$. Estimate $f(4)$ using a second degree interpolating polynomial (interpolating the first three data points) and a third degree interpolating polynomial (interpolating the first four data points). Round the final results to four decimal places. Is there any advantage here in using a third degree interpolating polynomial?

Python code for Newton interpolation

Consider the following finite difference table.

There are $2 + 1 = 3$ divided differences in the table, not counting the 0th divided differences. In general, the number of divided differences to compute is $1 + ... + n = n(n + 1)/2$. However, to construct Newton's form of the interpolating polynomial, we need only n divided differences and the 0th divided difference y_0. These numbers are displayed in red in Table 3.1. The important

x	$f(x)$	$f[x_i, x_{i+1}]$	$f[x_{i-1}, x_i, x_{i+1}]$
x_0	y_0		
		$\frac{y_1-y_0}{x_1-x_0} = f[x_0, x_1]$	
x_1	y_1		$\frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} = f[x_0, x_1, x_2]$
		$\frac{y_2-y_1}{x_2-x_1} = f[x_1, x_2]$	
x_2	y_2		

Table 3.1: Divided differences for three data points

observation is, even though all the divided differences have to be computed in order to get the ones needed for Newton's form, they do not have to be all stored. The following Python code is based on an efficient algorithm that goes through the divided difference calculations recursively, and stores an array of size $m = n + 1$ at any given time. In the final iteration, this array has the divided differences needed for Newton's form.

Let's explain the idea of the algorithm using the simple example of Table 3.1. The code creates an array $a = (a_0, a_1, a_2)$ of size $m = n + 1$, which is three in our example, and sets

$$a_0 = y_0, a_1 = y_1, a_2 = y_2.$$

In the first iteration (for loop, $j = 1$), a_1 and a_2 are updated:

$$a_1 := \frac{a_1 - a_0}{x_1 - x_0} = \frac{y_1 - y_0}{x_1 - x_0} = f[x_0, x_1], a_2 := \frac{a_2 - a_1}{x_2 - x_1} = \frac{y_2 - y_1}{x_2 - x_1}.$$

In the last iteration (for loop, $j = 2$), only a_2 is updated:

$$a_2 := \frac{a_2 - a_1}{x_2 - x_0} = \frac{\frac{y_2-y_1}{x_2-x_1} - \frac{y_1-y_0}{x_1-x_0}}{x_2 - x_0} = f[x_0, x_1, x_2].$$

The final array a is

$$a = (y_0, f[x_0, x_1], f[x_0, x_1, x_2])$$

containing the divided differences needed to construct the Newton's form of the polynomial.

Here is the Python function **diff** that computes the divided differences. The code uses a function we have not used before: **np.flip(np.arange(j,m))**. An example illustrates what it does the best:

```
In [1]: import numpy as np

In [2]: np.flip(np.arange(2,6))

Out[2]: array([5, 4, 3, 2])
```

In the code **diff** the inputs are the x- and y-coordinates of the data. The numbering of the indices starts at 0.

```
In [3]: def diff(x, y):
            m = x.size # here m is the number of data points.
            # the degree of the polynomial is m-1
            a = np.zeros(m)
            for i in range(m):
                a[i] = y[i]
            for j in range(1, m):
                for i in np.flip(np.arange(j,m)):
                    a[i] = (a[i]-a[i-1]) / (x[i]-x[i-(j)])
            return a
```

Let's compute the divided differences of Example 56:

```
In [4]: diff(np.array([-1,1,2]), np.array([-6,0,6]))

Out[4]: array([-6.,  3.,  1.])
```

These are the divided differences in the second ordering of the data in Example 57:

```
In [5]: diff(np.array([1.765,1.760,1.755,1.750]),
             np.array([0.92256,0.92137,0.92021,0.91906]))

Out[5]: array([ 0.92256   ,  0.238     ,  0.6       , 26.66666667])
```

Now let's write a code for the Newton form of polynomial interpolation. The inputs to the function **newton** are the x- and y-coordinates of the data, and where we want to evaluate the polynomial: z. The code uses the divided differences function **diff** discussed earlier to compute:

$$f[x_0] + f[x_0, x_1](z - x_0) + \ldots + f[x_0, x_1, ..., x_n](z - x_0)(z - x_1) \cdots (z - x_{n-1})$$

```
In [6]: def newton(x, y, z):
            m = x.size # here m is the number of data points, not the degree
            # of the polynomial
            a = diff(x, y)
            sum = a[0]
            pr = 1.0
            for j in range(m-1):
                pr *= (z-x[j])
                sum += a[j+1]*pr
            return sum
```

Let's verify the code by computing $p_3(1.761)$ of Example 57:

```
In [7]: newton(np.array([1.765,1.760,1.755,1.750]),
                np.array([0.92256,0.92137,0.92021,0.91906]), 1.761)

Out[7]: 0.92160496
```

Exercise 3.1-7: This problem discusses inverse interpolation which gives another method to find the root of a function. Let f be a continuous function on $[a, b]$ with one root p in the interval. Also assume f has an inverse. Let $x_0, x_1, ..., x_n$ be $n + 1$ distinct numbers in $[a, b]$ with $f(x_i) = y_i, i = 0, 1, ..., n$. Construct an interpolating polynomial P_n for $f^{-1}(x)$, by taking your data points as $(y_i, x_i), i = 0, 1, ..., n$. Observe that $f^{-1}(0) = p$, the root we are trying to find. Then, approximate the root p, by evaluating the interpolating polynomial for f^{-1} at 0, i.e., $P_n(0) \approx p$. Using this method, and the following data, find an approximation to the solution of $\log x = 0$.

x	0.4	0.8	1.2	1.6
$\log x$	-0.92	-0.22	0.18	0.47

3.2 High degree polynomial interpolation

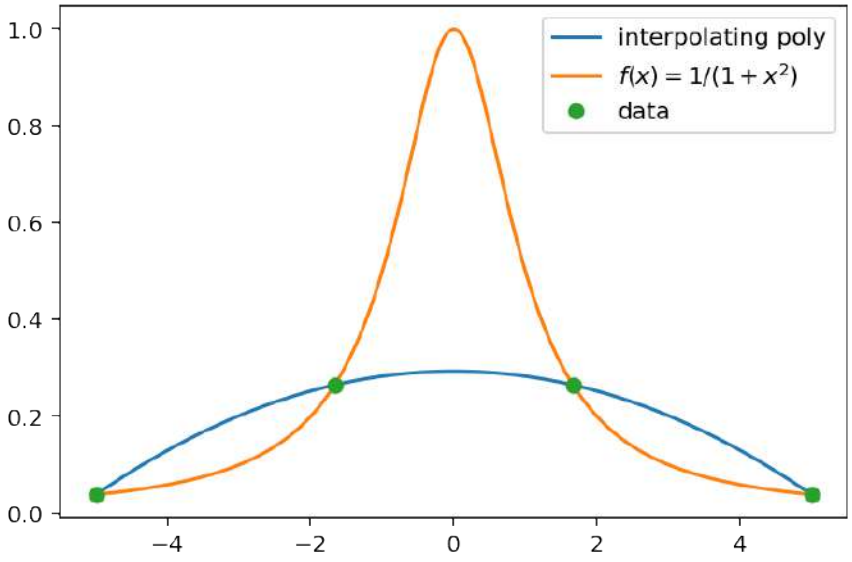
Suppose we approximate $f(x)$ using its polynomial interpolant $p_n(x)$ obtained from $(n + 1)$ data points. We then increase the number of data points, and update $p_n(x)$ accordingly. The central question we want to discuss is the following: as the number of nodes (data points) increases, does $p_n(x)$ become a better approximation to $f(x)$ on $[a, b]$? We will investigate this question numerically, using a famous example: Runge's function, given by $f(x) = \frac{1}{1+x^2}$.

We will interpolate Runge's function using polynomials of various degrees, and plot the function, together with its interpolating polynomial and the data points. We are interested to see what happens as the number of data points, and hence the degree of the interpolating polynomial, increases.

```
In [8]: import matplotlib.pyplot as plt
        %matplotlib inline
```

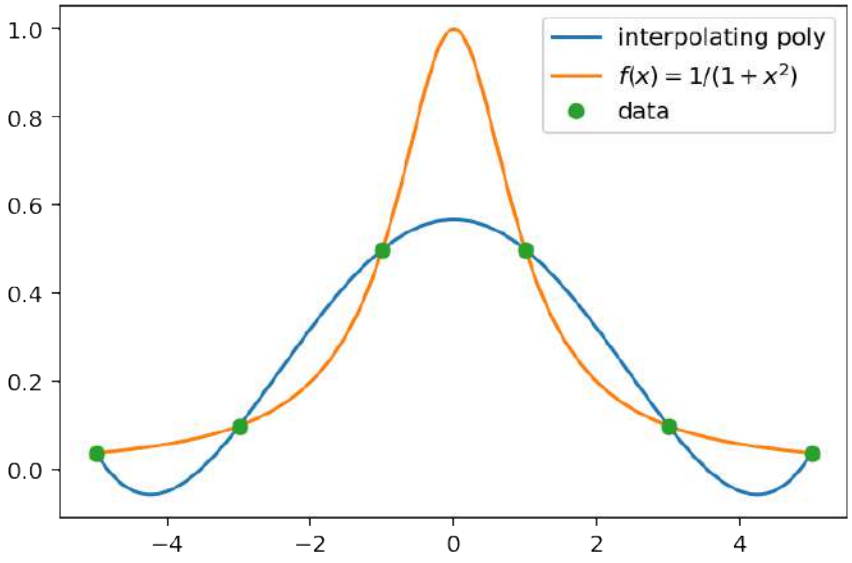
We start with taking four equally spaced x-coordinates between -5 and 5, and plot the corresponding interpolating polynomial and Runge's function. Matplotlib allows typing mathematics in captions of a plot using Latex. (Latex is a typesetting program this book is written with.) Latex commands need to be enclosed by a pair of dollar signs, in addition to a pair of quotation marks.

```
In [9]: f = lambda x: 1/(1+x**2)
        xi = np.linspace(-5, 5, 4) # x-coordinates of the data, 4 points equally spaced from
        #-5 to 5 in increments of 10/3
        yi = f(xi) # the corresponding y-coordinates
        xaxis = np.linspace(-5, 5, 1000)
        runge = f(xaxis) # Runge's function values
        interp = newton(xi, yi, xaxis)
        plt.plot(xaxis, interp, label='interpolating poly')
        plt.plot(xaxis, runge, label="$f(x)=1/(1+x^2)$")
        plt.plot(xi, yi, 'o', label='data')
        plt.legend(loc='upper right');
```



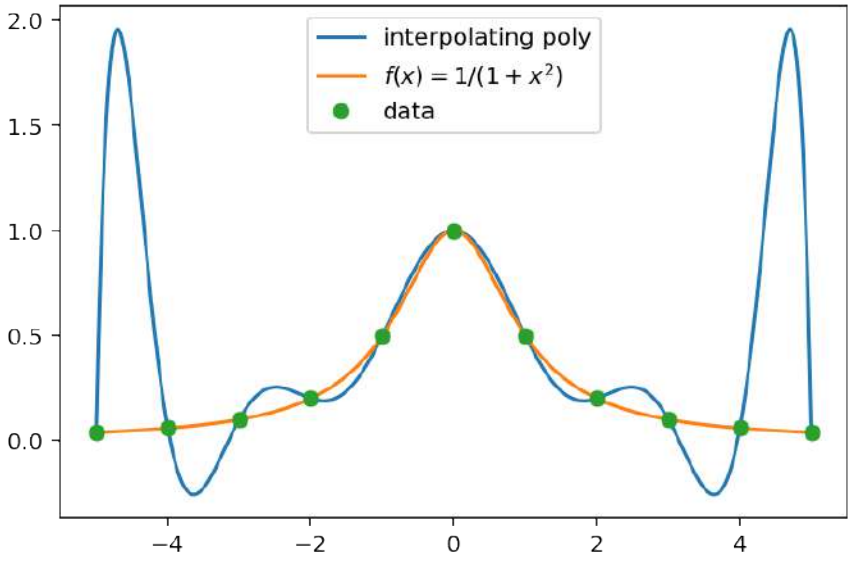
Next, we increase the number of data points to 6.

```
In [10]: xi = np.linspace(-5, 5, 6) # 6 equally spaced values from -5 to 5
         yi = f(xi) # the corresponding y-coordinates
         interp = newton(xi, yi, xaxis)
         plt.plot(xaxis, interp, label='interpolating poly')
         plt.plot(xaxis, runge, label="$f(x)=1/(1+x^2)$")
         plt.plot(xi, yi, 'o', label='data')
         plt.legend(loc='upper right');
```

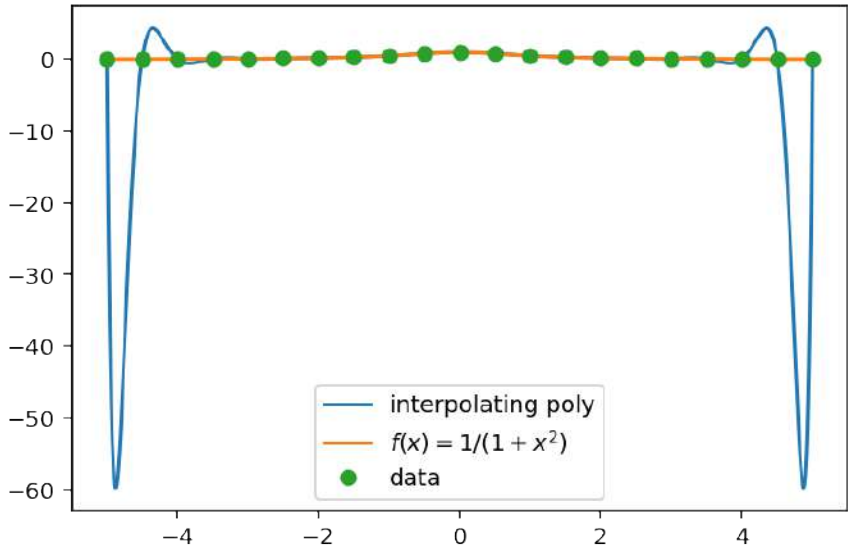


The next two graphs plot interpolating polynomials on 11 and 21 equally spaced data.

```
In [11]: xi = np.linspace(-5, 5, 11) # 11 equally spaced values from -5 to 5
         yi = f(xi) # the corresponding y-coordinates
         interp = newton(xi, yi, xaxis)
         plt.plot(xaxis, interp, label='interpolating poly')
         plt.plot(xaxis, runge, label="$f(x)=1/(1+x^2)$")
         plt.plot(xi, yi, 'o', label='data')
         plt.legend(loc='upper center');
```



```
In [12]: xi = np.linspace(-5, 5, 21) # 21 equally spaced values from -5 to 5
         yi = f(xi) # the corresponding y-coordinates
         interp = newton(xi, yi, xaxis)
         plt.plot(xaxis, interp, label='interpolating poly')
         plt.plot(xaxis, runge, label="$f(x)=1/(1+x^2)$")
         plt.plot(xi, yi, 'o', label='data')
         plt.legend(loc='lower center');
```



We observe that as the degree of the interpolating polynomial increases, the polynomial has large oscillations toward the end points of the interval. In fact, it can be shown that for any x such that $3.64 < |x| < 5, \sup_{n\geq 0} |f(x) - p_n(x)| = \infty$, where f is Runge's function.

This troublesome behavior of high degree interpolating polynomials improves significantly, if we consider data with x-coordinates that are **not** equally spaced. Consider the interpolation error of Theorem 51:

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n).$$

Perhaps surprisingly, the right-hand side of the above equation is not minimized when the nodes, x_i, are equally spaced! The set of nodes that minimizes the interpolation error is the roots of the so-called Chebyshev polynomials. The placing of these nodes is such that there are more nodes towards the end points of the interval, than the middle. We will learn about Chebyshev polynomials in Chapter 5. Using Chebyshev nodes in polynomial interpolation avoids the diverging behavior

of polynomial interpolants as the degree increases, as observed in the case of Runge's function, for sufficiently smooth functions.

Divided differences and derivatives

The following theorem shows the similarity between divided differences and derivatives.

Theorem 58. *Suppose $f \in C^n[a,b]$ and $x_0, x_1, ..., x_n$ are distinct numbers in $[a,b]$. Then there exists $\xi \in (a,b)$ such that*

$$f[x_0, ..., x_n] = \frac{f^{(n)}(\xi)}{n!}.$$

To prove this theorem, we need the generalized Rolle's theorem.

Theorem 59 (Rolle's theorem)**.** *Suppose f is a differentiable function on (a,b). If $f(a) = f(b)$, then there exists $c \in (a,b)$ such that $f'(c) = 0$.*

Theorem 60 (Generalized Rolle's theorem)**.** *Suppose f has n derivatives on (a,b). If $f(x) = 0$ at $(n+1)$ distinct numbers $x_0, x_1, ..., x_n \in [a,b]$, then there exists $c \in (a,b)$ such that $f^{(n)}(c) = 0$.*

Proof of Theorem 58 . Consider the function $g(x) = p_n(x) - f(x)$. Observe that $g(x_i) = 0$ for $i = 0, 1, ..., n$. From generalized Rolle's theorem, there exists $\xi \in (a,b)$ such that $g^{(n)}(\xi) = 0$, which implies

$$p_n^{(n)}(\xi) - f^{(n)}(\xi) = 0.$$

Since $p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + ... + f[x_0, ..., x_n](x - x_0) \cdots (x - x_{n-1})$, $p_n^{(n)}(x)$ equals $n!$ times the leading coefficient $f[x_0, ..., x_n]$. Therefore

$$f^{(n)}(\xi) = n! f[x_0, ..., x_n].$$

□

3.3 Hermite interpolation

In polynomial interpolation, our starting point has been the x- and y-coordinates of some data we want to interpolate. Suppose, in addition, we know the derivative of the underlying function at these x-coordinates. Our new data set has the following form.

Data:

$x_0, x_1, ..., x_n$

$y_0, y_1, ..., y_n;\ y_i = f(x_i)$

$$y'_0, y'_1, ..., y'_n;\ y'_i = f'(x_i)$$

We seek a polynomial that fits the y and y' values, that is, we seek a polynomial $H(x)$ such that $H(x_i) = y_i$ and $H'(x_i) = y'_i$, $i = 0, 1, ..., n$. This makes $2n + 2$ equations, and if we let

$$H(x) = a_0 + a_1 x + ... + a_{2n+1} x^{2n+1},$$

then there are $2n + 2$ unknowns, $a_0, ..., a_{2n+1}$, to solve for. The following theorem shows that there is a unique solution to this system of equations; a proof can be found in Burden, Faires, Burden [4].

Theorem 61. *If $f \in C^1[a, b]$ and $x_0, ..., x_n \in [a, b]$ are distinct, then there is a unique polynomial $H_{2n+1}(x)$, of degree at most $2n + 1$, agreeing with f and f' at $x_0, ..., x_n$. The polynomial can be written as:*

$$H_{2n+1}(x) = \sum_{i=0}^{n} y_i h_i(x) + \sum_{i=0}^{n} y'_i \widetilde{h}_i(x)$$

where

$$h_i(x) = \left(1 - 2(x - x_i) l'_i(x_i)\right) (l_i(x))^2$$
$$\widetilde{h}_i(x) = (x - x_i)(l_i(x))^2.$$

Here $l_i(x)$ is the ith Lagrange basis function for the nodes $x_0, ..., x_n$, and $l'_i(x)$ is its derivative. $H_{2n+1}(x)$ is called the Hermite interpolating polynomial.

The only difference between Hermite interpolation and polynomial interpolation is that in the former, we have the derivative information, which can go a long way in capturing the shape of the underlying function.

Example 62. We want to interpolate the following data:

$$x\text{-coordinates} : -1.5, 1.6, 4.7$$
$$y\text{-coordinates} : 0.071, -0.029, -0.012.$$

The underlying function the data comes from is $\cos x$, but we pretend we do not know this. Figure (3.2) plots the underlying function, the data, and the polynomial interpolant for the data. Clearly, the polynomial interpolant does not come close to giving a good approximation to the underlying function $\cos x$.

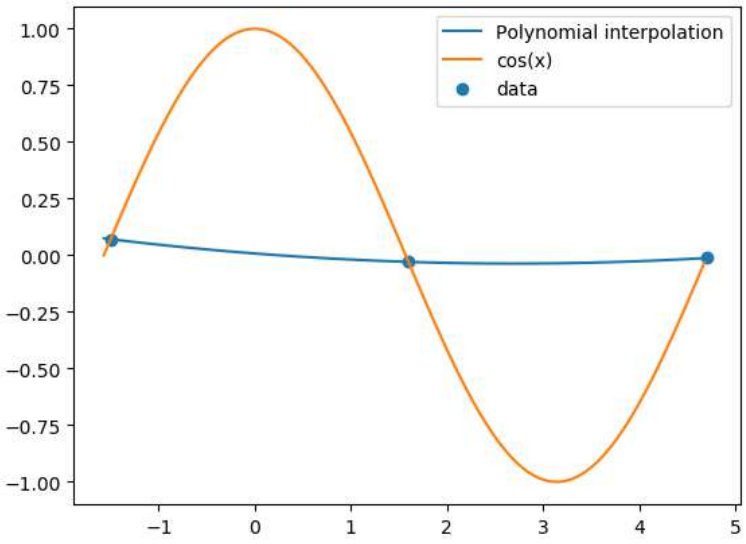


Figure 3.2

Now let's assume we know the derivative of the underlying function at these nodes:

$$x\text{-coordinates} : -1.5, 1.6, 4.7$$
$$y\text{-coordinates} : 0.071, -0.029, -0.012$$
$$y'\text{-values} : 1, -1, 1.$$

We then construct the Hermite interpolating polynomial, incorporating the derivative information. Figure (3.3) plots the Hermite interpolating polynomial, together with the polynomial interpolant, and the underlying function.

It is visually difficult to separate the Hermite interpolating polynomial from the underlying function $\cos x$ in Figure (3.3). Going from polynomial interpolation to Hermite interpolation results in rather dramatic improvement in approximating the underlying function.

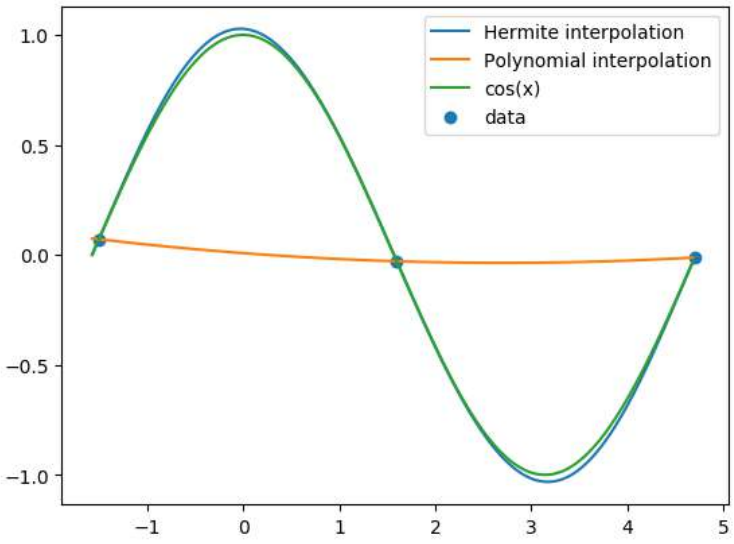


Figure 3.3

Computing the Hermite polynomial

We do not use Theorem 61 to compute the Hermite polynomial: there is a more efficient method using divided differences for this computation.

We start with the data:

$$x_0, x_1, ..., x_n$$
$$y_0, y_1, ..., y_n; y_i = f(x_i)$$
$$y'_0, y'_1, ..., y'_n; y'_i = f'(x_i)$$

and define a sequence $z_0, z_1, ..., z_{2n+1}$ by

$$z_0 = x_0, z_2 = x_1, z_4 = x_2, ..., z_{2n} = x_n$$
$$z_1 = x_0, z_3 = x_1, z_5 = x_2, ..., z_{2n+1} = x_n$$

i.e., $z_{2i} = z_{2i+1} = x_i$, for $i = 0, 1, ..., n$.

Then the Hermite polynomial can be written as:

$$\begin{aligned} H_{2n+1}(x) =& f[z_0] + f[z_0, z_1](x - z_0) + f[z_0, z_1, z_2](x - z_0)(x - z_1) + \\ & \ldots + f[z_0, z_1, ..., z_{2n+1}](x - z_0) \cdots (x - z_{2n}) \\ =& f[z_0] + \sum_{i=1}^{2n+1} f[z_0, ..., z_i](x - z_0)(x - z_1) \cdots (x - z_{i-1}). \end{aligned}$$

There is a little problem with some of the first divided differences above: they are undefined! Observe that

$$f[z_0, z_1] = f[x_0, x_0] = \frac{f(x_0) - f(x_0)}{x_0 - x_0}$$

or, in general,

$$f[z_{2i}, z_{2i+1}] = f[x_i, x_i] = \frac{f(x_i) - f(x_i)}{x_i - x_i}$$

for $i = 0, ..., n$.

From Theorem 58, we know $f[x_0, ..., x_n] = \frac{f^{(n)}(\xi)}{n!}$ for some ξ between the min and max of $x_0, ..., x_n$. From a classical result by Hermite & Gennochi (see Atkinson [3], page 144), divided differences are continuous functions of their variables $x_0, ..., x_n$. This implies we can take the limit of the above result as $x_i \to x_0$ for all i, which results in

$$f[x_0, ..., x_0] = \frac{f^{(n)}(x_0)}{n!}.$$

Therefore in the Hermite polynomial coefficient calculations, we will put

$$f[z_{2i}, z_{2i+1}] = f[x_i, x_i] = f'(x_i) = y_i'$$

for $i = 0, 1, ..., n$.

Example 63. Let's compute the Hermite polynomial of Example 62. The data is:

i	x_i	y_i	y_i'
0	-1.5	0.071	1
1	1.6	-0.029	-1
2	4.7	-0.012	1

Here $n = 2$, and $2n + 1 = 5$, so the Hermite polynomial is

$$H_5(x) = f[z_0] + \sum_{i=1}^{5} f[z_0, ..., z_i](x - z_0) \cdots (x - z_{i-1}).$$

The divided differences are:

z	$f(z)$	1st diff	2nd diff	3rd diff	4th diff	5th diff
$z_0 = -1.5$	0.071					
		$f'(z_0) = 1$				
$z_1 = -1.5$	0.071		$\frac{-0.032-1}{1.6+1.5} = -0.33$			
		$f[z_1, z_2] = -0.032$		0.0065		
$z_2 = 1.6$	-0.029		$\frac{-1+0.032}{1.6+1.5} = -0.31$		0.015	
		$f'(z_2) = -1$		0.10		-0.005
$z_3 = 1.6$	-0.029		$\frac{0.0055+1}{4.7-1.6} = 0.32$		-0.016	
		$f[z_3, z_4] = 0.0055$		0		
$z_4 = 4.7$	-0.012		$\frac{1-0.0055}{4.7-1.6} = 0.32$			
		$f'(z_4) = 1$				
$z_5 = 4.7$	-0.012					

Therefore, the Hermite polynomial is:

$$\begin{aligned} H_5(x) =& 0.071 + 1(x+1.5) - 0.33(x+1.5)^2 + 0.0065(x+1.5)^2(x-1.6) + \\ &+ 0.015(x+1.5)^2(x-1.6)^2 - 0.005(x+1.5)^2(x-1.6)^2(x-4.7). \end{aligned}$$

Python code for computing Hermite interpolating polynomial

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

The following function **hdiff** computes the divided differences needed for Hermite interpolation. It is based on the function **diff** for computing divided differences for Newton interpolation. The inputs to **hdiff** are the x-coordinates, the y-coordinates, and the derivatives **yprime**.

```
In [2]: def hdiff(x, y, yprime):
            m = x.size # here m is the number of data points. Note n=m-1
            # and 2n+1=2m-1
            l = 2*m
            z = np.zeros(l)
            a = np.zeros(l)
            for i in range(m):
                z[2*i] = x[i]
                z[2*i+1] = x[i]
            for i in range(m):
                a[2*i] = y[i]
```

```
        a[2*i+1] = y[i]
    for i in np.flip(np.arange(1, m)): # computes the first divided
        # differences using derivatives
        a[2*i+1] = yprime[i]
        a[2*i] = (a[2*i]-a[2*i-1]) / (z[2*i]-z[2*i-1])
    a[1] = yprime[0]
    for j in range(2, l): # computes the rest of the divided differences
        for i in np.flip(np.arange(j, l)):
            a[i]=(a[i]-a[i-1]) / (z[i]-z[i-j])
    return a
```

Let's compute the divided differences of Example 62.

```
In [3]: hdiff(np.array([-1.5, 1.6, 4.7]),
               np.array([0.071,-0.029,-0.012]),
               np.array([1,-1,1]))
```

```
Out[3]: array([ 0.071     ,  1.        , -0.33298647,  0.00671344,  0.0154761 ,
               -0.00519663])
```

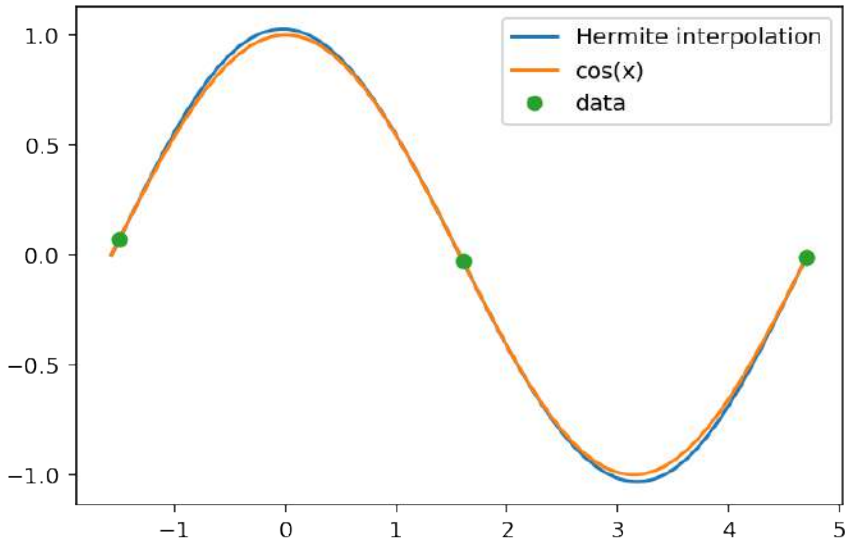
Note that in the hand-calculations of Example 63, where two-digit rounding was used, we obtained 0.0065 as the first third divided difference. In the Python output above, this divided difference is 0.0067.

The following function computes the Hermite interpolating polynomial, using the divided differences obtained from **hdiff**, and then evaluates the polynomial at w.

```
In [4]: def hermite(x, y, yprime, w):
            m = x.size # here m is the number of data points. not the
            # degree of the polynomial
            a = hdiff(x, y, yprime)
            z = np.zeros(2*m)
            for i in range(m):
                z[2*i] = x[i]
                z[2*i+1] = x[i]
            sum = a[0]
            pr = 1.0
            for j in range(2*m-1):
                pr *= w-z[j]
                sum += a[j+1]*pr
            return sum
```

Let's recreate the Hermite interpolating polynomial plot of Example 62.

```
In [5]: xaxis = np.linspace(-np.pi/2, 3*np.pi/2, 120)
        x = np.array([-1.5, 1.6, 4.7])
        y = np.array([0.071,-0.029,-0.012])
        yprime = np.array([1, -1, 1])
        funct = np.cos(xaxis)
        interp = hermite(x, y, yprime, xaxis)
        plt.plot(xaxis, interp, label='Hermite interpolation')
        plt.plot(xaxis, funct, label="cos(x)")
        plt.plot(x, y, 'o', label='data')
        plt.legend(loc='upper right');
```



Exercise 3.3-1: The following table gives the values of $y = f(x)$ and $y' = f'(x)$ where $f(x) = e^x + \sin 10x$. Compute the Hermite interpolating polynomial and the polynomial interpolant for the data in the table. Plot the two interpolating polynomials together with $f(x) = e^x + \sin 10x$ on $(0, 3)$.

x	0	0.4	1	2	2.6	3
y	1	0.735	2.17	8.30	14.2	19.1
y'	11	-5.04	-5.67	11.5	19.9	21.6

3.4 Piecewise polynomials: spline interpolation

As we observed in Section 3.2, a polynomial interpolant of high degree can have large oscillations, and thus provide an overall poor approximation to the underlying function. Recall that the degree of the interpolating polynomial is directly linked to the number of data points: we do not have the freedom to choose the degree of the polynomial.

In spline interpolation, we take a very different approach: instead of finding a single polynomial that fits the given data, we find one low-degree polynomial that fits every **pair** of data. This results in several polynomial pieces joined together, and we typically impose some smoothness conditions on different pieces. The term **spline function** means a function that consists of polynomial pieces joined together with some smoothness conditions.

In **linear spline interpolation**, we simply join data points (the nodes), by line segments, that is, linear polynomials. For example, consider the following figure that plots three data points $(x_{i-1}, y_{i-1}), (x_i, y_i), (x_{i+1}, y_{i+1})$. We fit a linear polynomial $P(x)$ to the first pair of data points $(x_{i-1}, y_{i-1}), (x_i, y_i)$, and another linear polynomial $Q(x)$ to the second pair of data points $(x_i, y_i), (x_{i+1}, y_{i+1})$.

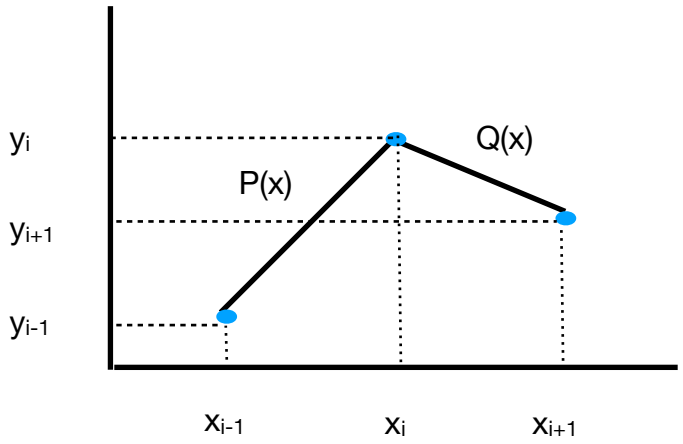


Figure 3.4: Linear spline

Let $P(x) = ax + b$ and $Q(x) = cx + d$. We find the coefficients a, b, c, d by solving

$$\begin{aligned} P(x_{i-1}) &= y_{i-1} \\ P(x_i) &= y_i \\ Q(x_i) &= y_i \\ Q(x_{i+1}) &= y_{i+1} \end{aligned}$$

which is a system of four equations and four unknowns. We then repeat this procedure for all data points, $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$, to determine all of the linear polynomials.

One disadvantage of linear spline interpolation is the lack of smoothness. The first derivative of the spline is not continuous at the nodes (unless the data fall on a line). We can obtain better smoothness by increasing the degree of the piecewise polynomials. In **quadratic spline interpolation**, we connect the nodes via second degree polynomials.

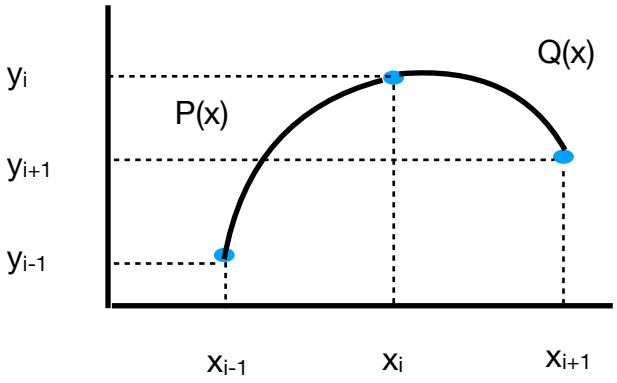


Figure 3.5: Quadratic spline

Let $P(x) = a_0 + a_1x + a_2x^2$ and $Q(x) = b_0 + b_1x + b_2x^2$. There are six unknowns to determine, but only four equations from the interpolation conditions: $P(x_{i-1}) = y_{i-1}, P(x_i) = y_i, Q(x_i) = y_i, Q(x_{i+1}) = y_{i+1}$. We can find extra two conditions by requiring some smoothness, $P'(x_i) = Q'(x_i)$, and another equation by requiring P' or Q' take a certain value at one of the end points.

Cubic spline interpolation

This is the most common spline interpolation. It uses cubic polynomials to connect the nodes. Consider the data

$$(x_0, y_0), (x_1, y_1), ..., (x_n, y_n),$$

where $x_0 < x_1 < ... < x_n$. In the figure below, the cubic polynomials interpolating pairs of data are labeled as $S_0, ..., S_{n-1}$ (we ignore the y-coordinates in the plot).

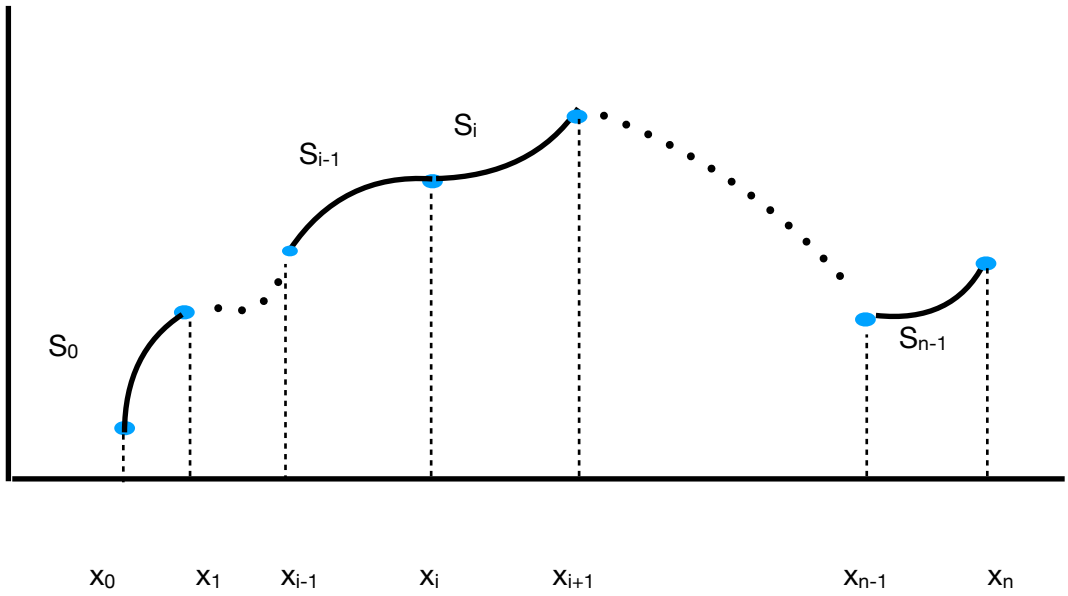


Figure 3.6: Cubic spline

The polynomial S_i interpolates the nodes $(x_i, y_i), (x_{i+1}, y_{i+1})$. Let

$$S_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$$

for $i = 0, 1, ..., n-1$. There are $4n$ unknowns to determine: a_i, b_i, c_i, d_i, as i takes on values from 0 to $n-1$. Let's describe the equations S_i must satisfy. First, the interpolation conditions, that is, the requirement that S_i passes through the nodes $(x_i, y_i), (x_{i+1}, y_{i+1})$:

$$S_i(x_i) = y_i$$
$$S_i(x_{i+1}) = y_{i+1}$$

for $i = 0, 1, ..., n-1$, which gives $2n$ equations. The next group of equations are about smoothness:

$$S'_{i-1}(x_i) = S'_i(x_i)$$
$$S''_{i-1}(x_i) = S''_i(x_i)$$

for $i = 1, 2, ..., n-1$, which gives $2(n-1) = 2n-2$ equations. Last two equations are called the boundary conditions. There are two choices:

- Free or natural boundary: $S''_0(x_0) = S''_{n-1}(x_n) = 0$
- Clamped boundary: $S'_0(x_0) = f'(x_0)$ and $S'_{n-1}(x_n) = f'(x_n)$

Each boundary choice gives another two equations, bringing the total number of equations to $4n$. There are $4n$ unknowns as well. Do these systems of equations have a unique solution? The answer is yes, and a proof can be found in Burden, Faires, Burden [4]. The spline obtained from the first boundary choice is called a **natural spline**, and the other one is called a **clamped spline**.

Example 64. Find the natural cubic spline that interpolates the data $(0,0),(1,1),(2,0)$.

Solution. We have two cubic polynomials to determine:

$$S_0(x) = a_0 + b_0x + c_0x^2 + d_0x^3$$
$$S_1(x) = a_1 + b_1x + c_1x^2 + d_1x^3$$

The interpolation equations are:

$$S_0(0) = 0 \Rightarrow a_0 = 0$$
$$S_0(1) = 1 \Rightarrow a_0 + b_0 + c_0 + d_0 = 1$$
$$S_1(1) = 1 \Rightarrow a_1 + b_1 + c_1 + d_1 = 1$$
$$S_1(2) = 0 \Rightarrow a_1 + 2b_1 + 4c_1 + 8d_1 = 0$$

We need the derivatives of the polynomials for the other equations:

$$S_0'(x) = b_0 + 2c_0x + 3d_0x^2$$
$$S_1'(x) = b_1 + 2c_1x + 3d_1x^2$$
$$S_0''(x) = 2c_0 + 6d_0x$$
$$S_1''(x) = 2c_1 + 6d_1x$$

The smoothness conditions are:

$$S_0'(1) = S_1'(1) \Rightarrow b_0 + 2c_0 + 3d_0 = b_1 + 2c_1 + 3d_1$$
$$S_0''(1) = S_1''(1) \Rightarrow 2c_0 + 6d_0 = 2c_1 + 6d_1$$

The natural boundary conditions are:

$$S_0''(0) = 0 \Rightarrow 2c_0 = 0$$
$$S_1''(2) = 0 \Rightarrow 2c_1 + 12d_1 = 0$$

There are eight equations and eight unknowns. However, $a_0 = c_0 = 0$, so that reduces the number of equations and unknowns to six. We rewrite the equations below, substituting $a_0 = c_0 = 0$, and

simplifying when possible:

$$\begin{aligned} b_0 + d_0 &= 1 \\ a_1 + b_1 + c_1 + d_1 &= 1 \\ a_1 + 2b_1 + 4c_1 + 8d_1 &= 0 \\ b_0 + 3d_0 &= b_1 + 2c_1 + 3d_1 \\ 3d_0 &= c_1 + 3d_1 \\ c_1 + 6d_1 &= 0 \end{aligned}$$

We will use Python to solve this system of equations. To do that, we first rewrite the system of equations as a matrix equation

$$Ax = v$$

where

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 & 4 & 8 \\ 1 & 3 & 0 & -1 & -2 & -3 \\ 0 & 3 & 0 & 0 & -1 & -3 \\ 0 & 0 & 0 & 0 & 1 & 6 \end{bmatrix}, x = \begin{bmatrix} b_0 \\ d_0 \\ a_1 \\ b_1 \\ c_1 \\ d_1 \end{bmatrix}, v = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

We enter the matrices A, v in Python and solve the equation $Ax = v$ using the command *np.linalg.solve*.

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

Out[1]: array([[ 1,  1,  0,  0,  0,  0],
               [ 0,  0,  1,  1,  1,  1],
               [ 0,  0,  1,  2,  4,  8],
               [ 1,  3,  0, -1, -2, -3],
               [ 0,  3,  0,  0, -1, -3],
               [ 0,  0,  0,  0,  1,  6]])

In [2]: v = np.array([1, 1, 0, 0, 0, 0])
        v

Out[2]: array([1, 1, 0, 0, 0, 0])

In [3]: np.linalg.solve(A, v)
```

```
Out[3]: array([ 1.5, -0.5, -1. ,  4.5, -3. ,  0.5])
```

Therefore, the polynomials are:

$$S_0(x) = 1.5x - 0.5x^3$$
$$S_1(x) = -1 + 4.5x - 3x^2 + 0.5x^3$$

Solving the equations of a spline even for three data points can be tedious. Fortunately, there is a general approach to solving the equations for natural and clamped splines, for any number of data points. We will use this approach when we write Python codes for splines next.

Exercise 3.4-1: Find the natural cubic spline interpolant for the following data:

x	-1	0	1
y	1	2	0

Exercise 3.4-2: The following is a clamped cubic spline for a function f defined on $[1, 3]$:

$$s(x) = \begin{cases} s_0(x) = (x-1) + (x-1)^2 - (x-1)^3, & \text{if } 1 \le x < 2 \\ s_1(x) = a + b(x-2) + c(x-2)^2 + d(x-2)^3 & \text{if } 2 \le x < 3. \end{cases}$$

Find a, b, c, and d, if $f'(1) = 1$ and $f'(3) = 2$.

Python code for spline interpolation

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

The function **CubicNatural** takes the x- and y-coordinates of the data as input, and computes the natural cubic spline interpolating the data, by solving the resulting matrix equation. The code is based on Algorithm 3.4 of Burden, Faires, Burden [4]. The output is the coefficients of the $m-1$ cubic polynomials, $a_i, b_i, c_i, d_i, i = 0, ..., m-2$ where m is the number of data points. These coefficients are stored in the arrays a, b, c, d and returned at the end of the function, so that we can access these arrays later to evaluate the spline for a given value w.

```
In [2]: def CubicNatural(x, y):
            m = x.size # m is the number of data points
            n = m-1
```

```
    a = np.zeros(m)
    b = np.zeros(n)
    c = np.zeros(m)
    d = np.zeros(n)
    for i in range(m):
        a[i] = y[i]
    h = np.zeros(n)
    for i in range(n):
        h[i] = x[i+1] - x[i]
    u = np.zeros(n)
    u[0] = 0
    for i in range(1, n):
        u[i] = 3*(a[i+1]-a[i])/h[i]-3*(a[i]-a[i-1])/h[i-1]
    s = np.zeros(m)
    z = np.zeros(m)
    t = np.zeros(n)
    s[0] = 1
    z[0] = 0
    t[0] = 0
    for i in range(1, n):
        s[i] = 2*(x[i+1]-x[i-1])-h[i-1]*t[i-1]
        t[i] = h[i]/s[i]
        z[i]=(u[i]-h[i-1]*z[i-1])/s[i]
    s[m-1] = 1
    z[m-1] = 0
    c[m-1] = 0
    for i in np.flip(np.arange(n)):
        c[i] = z[i]-t[i]*c[i+1]
        b[i] = (a[i+1]-a[i])/h[i]-h[i]*(c[i+1]+2*c[i])/3
        d[i] = (c[i+1]-c[i])/(3*h[i])
    return a, b, c, d
```

Once the matrix equation is solved, and the coefficients of the cubic polynomials are computed by **CubicNatural**, the next step is to evaluate the spline at a given value. This is done by the following function **CubicNaturalEval**. The inputs are the value at which the spline is evaluated, w, the x-coordinates of the data and the coefficients computed by **CubicNatural**. The function first finds the interval $[x_i, x_{i+1}], i = 0, ..., m-2$, w belongs to, and then evaluates the spline at w using the corresponding cubic polynomial.

```
In [3]: def CubicNaturalEval(w, x, coeff):
```

```
    m = x.size
    if w<x[0] or w>x[m-1]:
        print('error: spline evaluated outside its domain')
        return
    n = m-1
    p = 0
    for i in range(n):
        if w <= x[i+1]:
            break
        else:
            p += 1
    # p is the number of the subinterval w falls into, i.e., p=i means
    # w falls into the ith subinterval $(x_i,x_{i+1}), and therefore
    # the value of the spline at w is
    # a_i+b_i*(w-x_i)+c_i*(w-x_i)^2+d_i*(w-x_i)^3.
    a = coeff[0]
    b = coeff[1]
    c = coeff[2]
    d = coeff[3]
    return a[p]+b[p]*(w-x[p])+c[p]*(w-x[p])**2+d[p]*(w-x[p])**3
```

Next we will compare Newton and natural cubic spline interpolation when applied to Runge's function. We import the functions for Newton interpolation first.

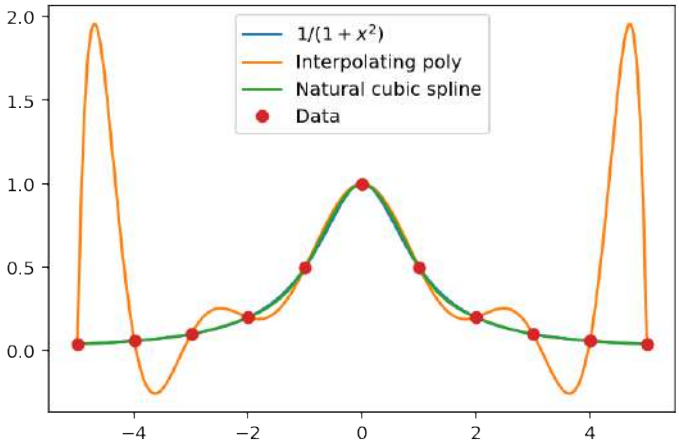
```
In [4]: def diff(x, y):
    m = x.size #here m is the number of data points.
    a = np.zeros(m)
    for i in range(m):
        a[i] = y[i]
    for j in range(1, m):
        for i in np.flip(np.arange(j,m)):
            a[i] = (a[i]-a[i-1]) / (x[i]-x[i-(j)])
    return a
```

```
In [5]: def newton(x, y, z):
    m = x.size # here m is the number of data points, not the degree
    # of the polynomial
    a = diff(x, y)
    sum = a[0]
    pr = 1.0
```

```
        for j in range(m-1):
            pr *= (z-x[j])
            sum += a[j+1]*pr
        return sum
```

Here is the code that computes the cubic spline, Newton interpolation, and plot them.

```
In [6]: xaxis = np.linspace(-5, 5, 1000)
        f = lambda x: 1/(1+x**2)
        runge = f(xaxis)
        xi = np.arange(-5, 6)
        yi = f(xi)
        coeff = CubicNatural(xi, yi)
        naturalspline = np.array(list(map(lambda x: CubicNaturalEval(x, xi, coeff), xaxis)))
        interp = newton(xi, yi, xaxis) # Interpolating polynomial for
        # the data
        plt.plot(xaxis, runge, label='$1/(1+x^2)$')
        plt.plot(xaxis, interp, label='Interpolating poly')
        plt.plot(xaxis, naturalspline, label='Natural cubic spline')
        plt.plot(xi, yi, 'o', label='Data')
        plt.legend(loc='upper center');
```



The cubic spline gives an excellent fit to Runge's function on this scale: we cannot visually separate it from the function itself.

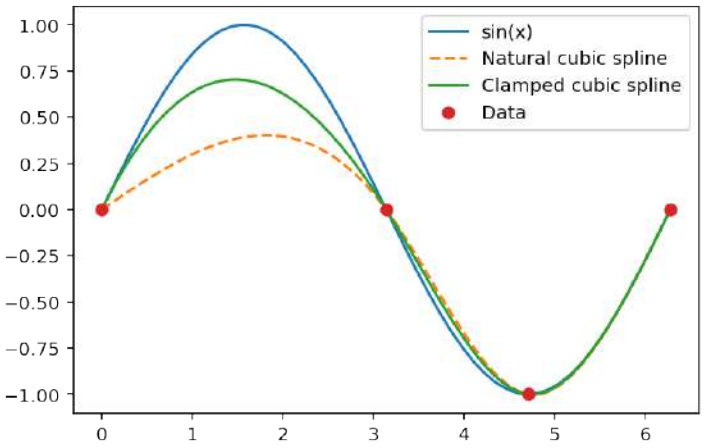
The following function **CubicClamped** computes the clamped cubic spline; the code is based on Algorithm 3.5 of Burden, Faires, Burden [4]. The function **CubicClampedEval** evaluates the spline at a given value.

```
In [7]: def CubicClamped(x, y, yprime_left, yprime_right):
            m = x.size # m is the number of data points
            n = m-1
            A = np.zeros(m)
            B = np.zeros(n)
            C = np.zeros(m)
            D = np.zeros(n)
            for i in range(m):
                A[i] = y[i]
            h = np.zeros(n)
            for i in range(n):
                h[i] = x[i+1] - x[i]
            u = np.zeros(m)
            u[0] = 3*(A[1]-A[0])/h[0]-3*yprime_left
            u[m-1] = 3*yprime_right-3*(A[m-1]-A[m-2])/h[m-2]
            for i in range(1, n):
                u[i] = 3*(A[i+1]-A[i])/h[i]-3*(A[i]-A[i-1])/h[i-1]
            s = np.zeros(m)
            z = np.zeros(m)
            t = np.zeros(n)
            s[0] = 2*h[0]
            t[0] = 0.5
            z[0] = u[0]/s[0]
            for i in range(1, n):
                s[i] = 2*(x[i+1]-x[i-1])-h[i-1]*t[i-1]
                t[i] = h[i]/s[i]
                z[i] = (u[i]-h[i-1]*z[i-1])/s[i]
            s[m-1] = h[m-2]*(2-t[m-2])
            z[m-1] = (u[m-1]-h[m-2]*z[m-2])/s[m-1]
            C[m-1] = z[m-1]
            for i in np.flip(np.arange(n)):
                C[i] = z[i]-t[i]*C[i+1]
                B[i] = (A[i+1]-A[i])/h[i]-h[i]*(C[i+1]+2*C[i])/3
                D[i] = (C[i+1]-C[i])/(3*h[i])
            return A, B, C, D

In [8]: def CubicClampedEval(w, x, coeff):
            m = x.size
            if w<x[0] or w>x[m-1]:
```

```
            print('error: spline evaluated outside its domain')
            return
        n = m-1
        p = 0
        for i in range(n):
            if w <= x[i+1]:
                break
            else:
                p += 1
        A = coeff[0]
        B = coeff[1]
        C = coeff[2]
        D = coeff[3]
        return A[p]+B[p]*(w-x[p])+C[p]*(w-x[p])**2+D[p]*(w-x[p])**3
```

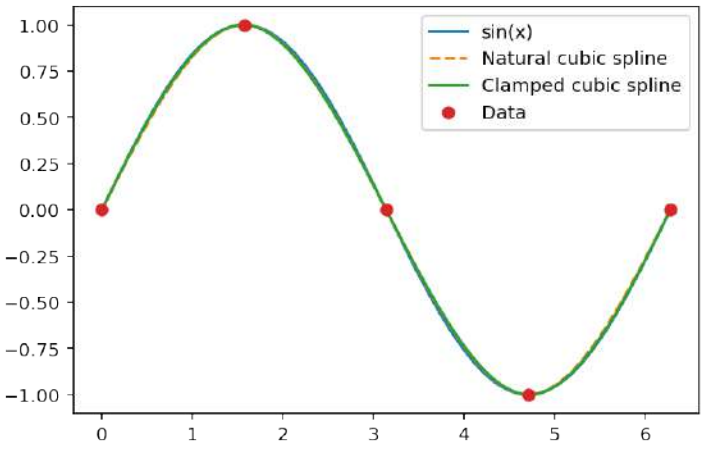
In the following, we use natural and clamped cubic splines to interpolate data coming from $\sin x$ at the x-coordinates: $0, \pi, 3\pi/2, 2\pi$. The derivatives at the end points are both equal to 1.

```
In [9]: xaxis = np.linspace(0, 2*np.pi, 600)
        f = lambda x: np.sin(x)
        funct = f(xaxis)
        xi = np.array([0, np.pi, 3*np.pi/2, 2*np.pi])
        yi = f(xi)
        coeff = CubicNatural(xi, yi)
        naturalspline = np.array(list(map(lambda x: CubicNaturalEval(x, xi, coeff), xaxis)))
        coeff = CubicClamped(xi, yi, 1, 1)
        clampedspline = np.array(list(map(lambda x: CubicClampedEval(x, xi, coeff), xaxis)))

        plt.plot(xaxis, funct, label='sin(x)')
        plt.plot(xaxis, naturalspline, linestyle='--', label='Natural cubic spline')
        plt.plot(xaxis, clampedspline, label='Clamped cubic spline')
        plt.plot(xi, yi, 'o', label='Data')
        plt.legend(loc='upper right');
```



Especially on the interval $(0, \pi)$, the clamped spline gives a much better approximation to $\sin x$ than the natural spline. However, adding an extra data point between 0 and π removes the visual differences between the splines.

```
In [10]: xaxis = np.linspace(0, 2*np.pi, 600)
         f = lambda x: np.sin(x)
         funct = f(xaxis)
         xi = np.array([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi])
         yi = f(xi)
         coeff = CubicNatural(xi, yi)
         naturalspline = np.array(list(map(lambda x: CubicNaturalEval(x, xi, coeff), xaxis)))
         coeff = CubicClamped(xi, yi, 1, 1)
         clampedspline = np.array(list(map(lambda x: CubicClampedEval(x, xi, coeff), xaxis)))

         plt.plot(xaxis, funct, label='sin(x)')
         plt.plot(xaxis, naturalspline, linestyle='--', label='Natural cubic spline')
         plt.plot(xaxis, clampedspline, label='Clamped cubic spline')
         plt.plot(xi, yi, 'o', label='Data')
         plt.legend(loc='upper right');
```



Arya and the letter NUH

Arya loves Dr. Seuss (who doesn't?), and she is writing her term paper in an English class on *On Beyond Zebra!*[2]. In this book Dr. Seuss invents new letters, one of which is called NUH. He writes:

> *And NUH is the letter I use to spell Nutches*
> *Who live in small caves, known as Nitches, for hutches.*
> *These Nutches have troubles, the biggest of which is*
> *The fact there are many more Nutches than Nitches.*

What does this letter look like? Well, something like this.

What Arya wants is a digitized version of the sketch; a figure that is smooth and can be manipulated using graphics software. The letter NUH is a little complicated to apply a spline interpolation directly, since it has some cusps. For such planar curves, we can use their parametric representation, and use a cubic spline interpolation for x- and y-coordinates separately. To this end, Arya picks eight points on the letter NUH, and labels them as $t = 1, 2, ..., 8$; see the figure below.

[2]Seuss, 1955. On Beyond Zebra! Random House for Young Readers.

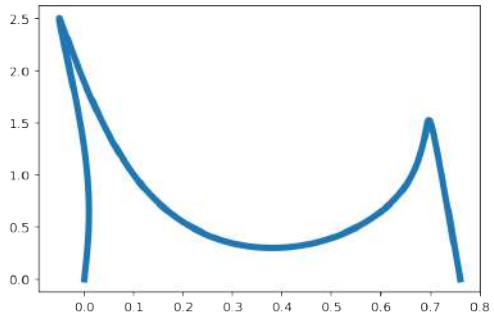
Then for each point she eyeballs the x and y-coordinates with the help of a graph paper. The results are displayed in the table below.

t	1	2	3	4	5	6	7	8
x	0	0	-0.05	0.1	0.4	0.65	0.7	0.76
y	0	1.25	2.5	1	0.3	0.9	1.5	0

The next step is to fit a cubic spline to the data $(t_1, x_1), ..., (t_8, x_8)$, and another cubic spline to the data $(t_1, y_1), ..., (t_8, y_8)$. Let's call these splines xspline(t), yspline(t), respectively, since they represent the x- and y-coordinates as functions of the parameter t. Plotting xspline(t), yspline(t) will produce the letter NUH, as we can see in the following Python codes.

First, load the NumPy and Matplotlib packages, and copy and evaluate the functions **CubicNatural** and **CubicNaturalEval** that we discussed earlier. Here is the letter NUH, obtained by spline interpolation:

```
In [1]: t = np.array([1,2,3,4,5,6,7,8])
        x = np.array([0,0,-0.05,0.1,0.4,0.65,0.7,0.76])
        y = np.array([0,1.25,2.5,1,0.3,0.9,1.5,0])
        taxis = np.linspace(1, 8, 700)
        coeff = CubicNatural(t, x)
        xspline = np.array(list(map(lambda x: CubicNaturalEval(x, t, coeff), taxis)))

        coeff = CubicNatural(t, y)
        yspline = np.array(list(map(lambda x: CubicNaturalEval(x, t, coeff), taxis)))

        plt.plot(xspline, yspline, linewidth=5);
```

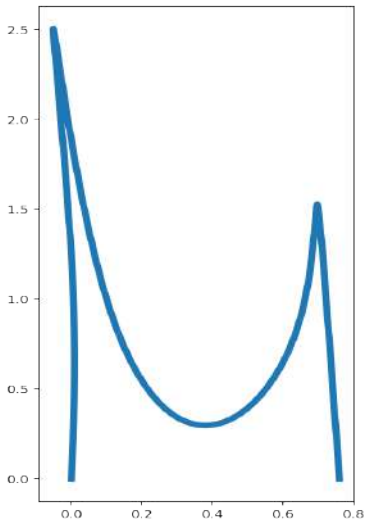


This looks like it needs to be squeezed! Adjusting the aspect ratio gives a better image. In the following, we use the commands

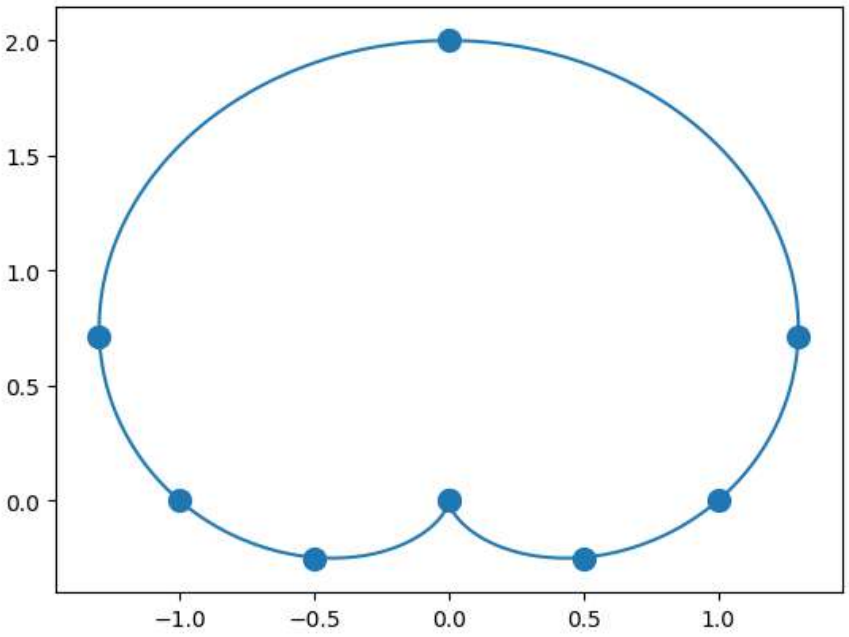
```
In [2]: w, h = plt.figaspect(2);
        plt.figure(figsize=(w, h));
```

to adjust the aspect ratio.

```
In [3]: t = np.array([1,2,3,4,5,6,7,8])
        x = np.array([0,0,-0.05,0.1,0.4,0.65,0.7,0.76])
        y = np.array([0,1.25,2.5,1,0.3,0.9,1.5,0])
        taxis = np.linspace(1, 8, 700)
        coeff = CubicNatural(t, x)
        xspline = np.array(list(map(lambda x: CubicNaturalEval(x, t, coeff), taxis)))

        coeff = CubicNatural(t, y)
        yspline = np.array(list(map(lambda x: CubicNaturalEval(x, t, coeff), taxis)))

        w, h = plt.figaspect(2)
        plt.figure(figsize=(w, h))
        plt.plot(xspline, yspline, linewidth=5);
```



Exercise 3.4-3: Limaçon is a curve, named after a French word for snail, which appears in the study of planetary motion. The polar equation for the curve is $r = 1 + c\sin\theta$ where c is a constant. Below is a plot of the curve when $c = 1$.



The x, y coordinates of the dots on the curve are displayed in the following table:

x	0	0.5	1	1.3	0	-1.3	-1	-0.5	0
y	0	-0.25	0	0.71	2	0.71	0	-0.25	0

Recreate the limaçon above, by applying the spline interpolation for plane curves approach used in *Arya and the letter NUH* example to the points given in the table.

Chapter 4

Numerical Quadrature and Differentiation

Estimating $\int_a^b f(x)dx$ using sums of the form $\sum_{i=0}^n w_i f(x_i)$ is known as the quadrature problem. Here w_i are called weights, and x_i are called nodes. The objective is to determine the nodes and weights to minimize error.

4.1 Newton-Cotes formulas

The idea is to construct the polynomial interpolant $P(x)$ and compute $\int_a^b P(x)dx$ as an approximation to $\int_a^b f(x)dx$. Given nodes $x_0, x_1, ..., x_n$, the Lagrange form of the interpolant is

$$P_n(x) = \sum_{i=0}^{n} f(x_i) l_i(x)$$

and from the interpolation error formula Theorem 51, we have

$$f(x) = P_n(x) + (x - x_0) \cdots (x - x_n) \frac{f^{(n+1)}(\xi(x))}{(n+1)!},$$

where $\xi(x) \in [a, b]$. (We have written $\xi(x)$ instead of ξ to emphasize that ξ depends on the value of x.)

Taking the integral of both sides yields

$$\int_a^b f(x)dx = \underbrace{\int_a^b P_n(x)dx}_{\text{quadrature rule}} + \underbrace{\frac{1}{(n+1)!} \int_a^b \prod_{i=0}^{n} (x - x_i) f^{(n+1)}(\xi(x))dx.}_{\text{error term}} \tag{4.1}$$

The first integral on the right-hand side gives the quadrature rule:

$$\int_a^b P_n(x)dx = \int_a^b \left(\sum_{i=0}^{n} f(x_i)l_i(x)\right) dx = \sum_{i=0}^{n} \left(\underbrace{\int_a^b l_i(x)dx}_{w_i}\right) f(x_i) = \sum_{i=0}^{n} w_i f(x_i),$$

and the second integral gives the error term.

We obtain different quadrature rules by taking different nodes, or number of nodes. The following result is useful in the theoretical analysis of Newton-Cotes formulas.

Theorem 65 (Weighted mean value theorem for integrals)**.** *Suppose $f \in C^0[a,b]$, the Riemann integral of $g(x)$ exists on $[a,b]$, and $g(x)$ does not change sign on $[a,b]$. Then there exists $\xi \in (a,b)$ with $\int_a^b f(x)g(x)dx = f(\xi)\int_a^b g(x)dx$.*

Two well-known numerical quadrature rules, trapezoidal rule and Simpson's rule, are examples of Newton-Cotes formulas:

- **Trapezoidal rule**

  Let $f \in C^2[a,b]$. Take two nodes, $x_0 = a, x_1 = b$, and use the linear Lagrange polynomial

  $$P_1(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1)$$

  to estimate $f(x)$. Substitute $n = 1$ in Equation (4.1) to get

  $$\int_a^b f(x)dx = \int_a^b P_1(x)dx + \frac{1}{2}\int_a^b \prod_{i=0}^{1}(x - x_i) f''(\xi(x))dx,$$

  and then substitute for P_1 to obtain

  $$\int_a^b f(x)dx = \int_a^b \frac{x - x_1}{x_0 - x_1} f(x_0)dx + \int_a^b \frac{x - x_0}{x_1 - x_0} f(x_1)dx + \frac{1}{2}\int_a^b (x - x_0)(x - x_1) f''(\xi(x))dx.$$

  The first two integrals on the right-hand side are $\frac{h}{2}f(x_0)$ and $\frac{h}{2}f(x_1)$. Let's evaluate

  $$\int_a^b (x - x_0)(x - x_1) f''(\xi(x))dx.$$

  We will use Theorem 65 for this computation. Note that the function $(x - x_0)(x - x_1) = (x - a)(x - b)$ does not change sign on the interval $[a,b]$ and it is integrable: so this function serves the role of $g(x)$ in Theorem 65. The other term, $f''(\xi(x))$, serves the role of $f(x)$. Applying the theorem, we get

  $$\int_a^b (x - a)(x - b) f''(\xi(x))dx = f''(\xi)\int_a^b (x - a)(x - b)dx$$

where we kept the same notation ξ, somewhat inappropriately, as we moved $f''(\xi(x))$ from inside to the outside of the integral. Finally, observe that

$$\int_a^b (x-a)(x-b)dx = \frac{(a-b)^3}{6} = \frac{-h^3}{6},$$

where $h = b - a$. Putting all the pieces together, we have obtained

$$\int_a^b f(x)dx = \frac{h}{2}\left[f(x_0) + f(x_1)\right] - \frac{h^3}{12}f''(\xi).$$

- **Simpson's rule**

  Let $f \in C^4[a,b]$. Take three equally-spaced nodes, $x_0 = a$, $x_1 = a+h$, $x_2 = b$, where $h = \frac{b-a}{2}$, and use the second degree Lagrange polynomial

$$P_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}f(x_2)$$

  to estimate $f(x)$. Substitute $n = 2$ in Equation (4.1) to get

$$\int_a^b f(x)dx = \int_a^b P_2(x)dx + \frac{1}{3!}\int_a^b \prod_{i=0}^{2}(x-x_i)f^{(3)}(\xi(x))dx,$$

  and then substitute for P_2 to obtain

$$\begin{aligned}\int_a^b f(x)dx = &\int_a^b \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}f(x_0)dx + \int_a^b \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}f(x_1)dx + \\ &+ \int_a^b \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}f(x_2)dx + \frac{1}{6}\int_a^b (x-x_0)(x-x_1)(x-x_2)f^{(3)}(\xi(x))dx.\end{aligned}$$

  The sum of the first three integrals on the right-hand side simplify as: $\frac{h}{3}\left[f(x_0) + 4f(x_1) + f(x_2)\right]$. The last integral cannot be evaluated using Theorem 65 directly, like in the trapezoidal rule, since the function $(x-x_0)(x-x_1)(x-x_2)$ changes sign on $[a,b]$. However, a clever application of integration by parts transforms the integral to an integral where Theorem 65 is applicable (see Atkinson [3] for details), and the integral simplifies as $-\frac{h^5}{90}f^{(4)}(\xi)$ for some $\xi \in (a,b)$. In summary, we obtain

$$\int_a^b f(x)dx = \frac{h}{3}\left[f(x_0) + 4f(x_1) + f(x_2)\right] - \frac{h^5}{90}f^{(4)}(\xi)$$

  where $\xi \in (a,b)$.

Exercise 4.1-1: Prove that the sum of the weights in Newton-Cotes rules is $b-a$, for any n.

Definition 66. The degree of accuracy, or precision, of a quadrature formula is the largest positive integer n such that the formula is exact for $f(x) = x^k$, when $k = 0, 1, ..., n$, or equivalently, for any polynomial of degree less than or equal to n.

Observe that the trapezoidal and Simpson's rules have degrees of accuracy of one and three. These two rules are examples of closed Newton-Cotes formulas; closed refers to the fact that the end points a, b of the interval are used as nodes in the quadrature rule. Here is the general definition.

Definition 67 (Closed Newton-Cotes)**.** The $(n+1)$-point closed Newton-Cotes formula uses nodes $x_i = x_0 + ih$, for $i = 0, 1, ..., n$, where $x_0 = a, x_n = b, h = \frac{b-a}{n}$, and

$$w_i = \int_{x_0}^{x_n} l_i(x)dx = \int_{x_0}^{x_n} \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} dx.$$

The following theorem provides an error formula for the closed Newton-Cotes formula. A proof can be found in Isaacson and Keller [12].

Theorem 68. *For the $(n+1)$-point closed Newton-Cotes formula, we have:*

- *if n is even and $f \in C^{n+2}[a, b]$*

$$\int_a^b f(x)dx = \sum_{i=0}^{n} w_i f(x_i) + \frac{h^{n+3} f^{(n+2)}(\xi)}{(n+2)!} \int_0^n t^2(t-1)\cdots(t-n)dt,$$

- *if n is odd and $f \in C^{n+1}[a, b]$*

$$\int_a^b f(x)dx = \sum_{i=0}^{n} w_i f(x_i) + \frac{h^{n+2} f^{(n+1)}(\xi)}{(n+1)!} \int_0^n t(t-1)\cdots(t-n)dt$$

where ξ is some number in (a, b).

Some well known examples of closed Newton-Cotes formulas are the trapezoidal rule ($n = 1$), Simpson's rule ($n = 2$), and Simpson's three-eighth rule ($n = 3$). Observe that in the $(n+1)$-point closed Newton-Cotes formula, if n is even, then the degree of accuracy is $(n+1)$, although the interpolating polynomial is of degree n. The open Newton-Cotes formulas exclude the end points of the interval.

Definition 69 (Open Newton-Cotes)**.** The $(n+1)$-point open Newton-Cotes formula uses nodes $x_i = x_0 + ih$, for $i = 0, 1, ..., n$, where $x_0 = a + h, x_n = b - h, h = \frac{b-a}{n+2}$ and

$$w_i = \int_a^b l_i(x)dx = \int_a^b \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} dx.$$

We put $a = x_{-1}$ and $b = x_{n+1}$.

The error formula for the open Newton-Cotes formula is given next; for a proof see Isaacson and Keller [12].

Theorem 70. *For the $(n+1)$-point open Newton-Cotes formula, we have:*

- *if n is even and $f \in C^{n+2}[a,b]$*

$$\int_a^b f(x)dx = \sum_{i=0}^{n} w_i f(x_i) + \frac{h^{n+3} f^{(n+2)}(\xi)}{(n+2)!} \int_{-1}^{n+1} t^2(t-1)\cdots(t-n)dt,$$

- *if n is odd and $f \in C^{n+1}[a,b]$*

$$\int_a^b f(x)dx = \sum_{i=0}^{n} w_i f(x_i) + \frac{h^{n+2} f^{(n+1)}(\xi)}{(n+1)!} \int_{-1}^{n+1} t(t-1)\cdots(t-n)dt,$$

  where ξ is some number in (a,b).

The well-known midpoint rule is an example of open Newton-Cotes formula:

- **Midpoint rule**
  Take one node, $x_0 = a + h$, which corresponds to $n = 0$ in the above theorem to obtain

$$\int_a^b f(x)dx = 2hf(x_0) + \frac{h^3 f''(\xi)}{3}$$

  where $h = (b-a)/2$. This rule interpolates f by a constant (the values of f at the midpoint), that is, a polynomial of degree 0, but it has degree of accuracy 1.

Remark 71. Both closed and open Newton-Cotes formulas using odd number of nodes (n is even), gain an extra degree of accuracy beyond that of the polynomial interpolant on which they are based. This is due to cancellation of positive and negative error.

There are some drawbacks of Newton-Cotes formulas:

- In general, these rules are not of the highest degree of accuracy possible for the number of nodes used.
- The use of large number of equally spaced nodes may incur the erratic behavior associated with high-degree polynomial interpolation. Weights for a high-order rule may be negative, potentially leading to loss of significance errors.
- Let I_n denote the Newton-Cotes estimate of an integral based on n nodes. I_n may not converge to the true integral as $n \to \infty$ for perfectly well-behaved integrands.

Example 72. Estimate $\int_{0.5}^{1} x^x dx$ using the midpoint, trapezoidal, and Simpson's rules.

Solution. Let $f(x) = x^x$. The midpoint estimate for the integral is $2hf(x_0)$ where $h = (b-a)/2 = 1/4$ and $x_0 = 0.75$. Then the midpoint estimate, using 6 digits, is $f(0.75)/2 = 0.805927/2 = 0.402964$. The trapezoidal estimate is $\frac{h}{2}[f(0.5) + f(1)]$ where $h = 1/2$, which results in $1.707107/4 = 0.426777$. Finally, for Simpson's rule, $h = (b-a)/2 = 1/4$, and thus the estimate is

$$\frac{h}{3}[f(0.5) + 4f(0.75) + f(1)] = \frac{1}{12}[0.707107 + 4(0.805927) + 1] = 0.410901.$$

Here is a summary of the results:

Midpoint	Trapezoidal	Simpson's
0.402964	0.426777	0.410901

Example 73. Find the constants c_0, c_1, x_1 so that the quadrature formula

$$\int_0^1 f(x)dx = c_0 f(0) + c_1 f(x_1)$$

has the highest possible degree of accuracy.

Solution. We will find how many of the polynomials $1, x, x^2, ...$ the rule can integrate exactly. If $p(x) = 1$, then

$$\int_0^1 p(x)dx = c_0 p(0) + c_1 p(x_1) \Rightarrow 1 = c_0 + c_1.$$

If $p(x) = x$, we get

$$\int_0^1 p(x)dx = c_0 p(0) + c_1 p(x_1) \Rightarrow \frac{1}{2} = c_1 x_1$$

and $p(x) = x^2$ implies

$$\int_0^1 p(x)dx = c_0 p(0) + c_1 p(x_1) \Rightarrow \frac{1}{3} = c_1 x_1^2.$$

We have three unknowns and three equations, so we have to stop here. Solving the three equations we get: $c_0 = 1/4, c_1 = 3/4, x_1 = 2/3$. So the quadrature rule is of precision two and it is:

$$\int_0^1 f(x)dx = \frac{1}{4}f(0) + \frac{3}{4}f(\frac{2}{3}).$$

Exercise 4.1-2: Find c_0, c_1, c_2 so that the quadrature rule $\int_{-1}^1 f(x)dx = c_0 f(-1) + c_1 f(0) + c_2 f(1)$ has degree of accuracy 2.

4.2 Composite Newton-Cotes formulas

If the interval $[a, b]$ in the quadrature is large, then the Newton-Cotes formulas will give poor approximations. The quadrature error depends on $h = (b - a)/n$ (closed formulas), and if $b - a$ is large, then so is h, hence error. If we raise n to compensate for large interval, then we face a problem discussed earlier: error due to the oscillatory behavior of high-degree interpolating polynomials that use equally-spaced nodes. A solution is to break up the domain into smaller intervals and use a Newton-Cotes rule with a smaller n on each subinterval: this is known as a composite rule.

Example 74. Let's compute $\int_0^2 e^x \sin x dx$. The antiderivative can be computed using integration by parts, and the true value of the integral to 6 digits is 5.39689. If we apply the Simpson's rule we get:

$$\int_0^2 e^x \sin x dx \approx \frac{1}{3}(e^0 \sin 0 + 4e \sin 1 + e^2 \sin 2) = 5.28942.$$

If we partition the integration domain $(0, 2)$ into $(0, 1)$ and $(1, 2)$, and apply Simpson's rule to each domain separately, we get

$$\begin{aligned}\int_0^2 e^x \sin x dx &= \int_0^1 e^x \sin x dx + \int_1^2 e^x \sin x dx \\ &\approx \frac{1}{6}(e^0 \sin 0 + 4e^{0.5} \sin(0.5) + e \sin 1) + \frac{1}{6}(e \sin 1 + 4e^{1.5} \sin(1.5) + e^2 \sin 2) \\ &= 5.38953,\end{aligned}$$

improving the accuracy significantly. Note that we have used five nodes, $0, 0.5, 1, 1.5, 2$, which split the domain $(0, 2)$ into four subintervals.

The composite rules for midpoint, trapezoidal, and Simpson's rule, with their error terms, are:

- **Composite Midpoint rule**
  Let $f \in C^2[a, b]$, n be even, $h = \frac{b-a}{n+2}$, and $x_j = a + (j + 1)h$ for $j = -1, 0, ..., n + 1$. The composite Midpoint rule for $n + 2$ subintervals is

$$\int_a^b f(x)dx = 2h \sum_{j=0}^{n/2} f(x_{2j}) + \frac{b-a}{6} h^2 f''(\xi) \tag{4.2}$$

  for some $\xi \in (a, b)$.

- **Composite Trapezoidal rule**
  Let $f \in C^2[a, b]$, $h = \frac{b-a}{n}$, and $x_j = a + jh$ for $j = 0, 1, ..., n$. The composite Trapezoidal rule for n subintervals is

$$\int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right] - \frac{b-a}{12} h^2 f''(\xi) \tag{4.3}$$

for some $\xi \in (a, b)$.

- **Composite Simpson's rule**
  Let $f \in C^4[a, b]$, n be even, $h = \frac{b-a}{n}$, and $x_j = a + jh$ for $j = 0, 1, ..., n$. The composite Simpson's rule for n subintervals is

$$\int_a^b f(x)dx = \frac{h}{3}\left[f(a) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(b)\right] - \frac{b-a}{180}h^4 f^{(4)}(\xi) \tag{4.4}$$

  for some $\xi \in (a, b)$.

Exercise 4.2-1: Show that the quadrature rule in Example 74 corresponds to taking $n = 4$ in the composite Simpson's formula (4.4).

Exercise 4.2-2: Show that the absolute error for the composite trapezoidal rule decays at the rate of $1/n^2$, and the absolute error for the composite Simpson's rule decays at the rate of $1/n^4$, where n is the number of subintervals.

Example 75. Determine n that ensures the composite Simpson's rule approximates $\int_1^2 x \log x dx$ with an absolute error of at most 10^{-6}.

Solution. The error term for the composite Simpson's rule is $\frac{b-a}{180}h^4 f^{(4)}(\xi)$ where ξ is some number between $a = 1$ and $b = 2$, and $h = (b - a)/n$. Differentiate to get $f^{(4)}(x) = \frac{2}{x^3}$. Then

$$\frac{b-a}{180}h^4 f^{(4)}(\xi) = \frac{1}{180}h^4\frac{2}{\xi^3} \leq \frac{h^4}{90}$$

where we used the fact that $\frac{2}{\xi^3} \leq \frac{2}{1} = 2$ when $\xi \in (1, 2)$. Now make the upper bound less than 10^{-6}, that is,

$$\frac{h^4}{90} \leq 10^{-6} \Rightarrow \frac{1}{n^4(90)} \leq 10^{-6} \Rightarrow n^4 \geq \frac{10^6}{90} \approx 11111.11$$

which implies $n \geq 10.27$. Since n must be even for Simpson's rule, this means the smallest value of n to ensure an error of at most 10^{-6} is 12.

Using the Python code for the composite Simpson's rule that will be introduced next, we get 0.6362945608 as the estimate, using 10 digits. The correct integral to 10 digits is 0.6362943611, which means an absolute error of 2×10^{-7}, better than the expected 10^{-6}.

Python codes for Newton-Cotes formulas

We write codes for the trapezoidal and Simpson's rules, and the composite Simpson's rule. Coding trapezoidal and Simpson's rule is straightforward.

Trapezoidal rule

```
In [1]: def trap(f, a, b):
            return (f(a)+f(b))*(b-a)/2
```

Let's verify the calculations of Example 72:

```
In [2]: trap(x->x^x,0.5,1)

Out[2]: 0.42677669529663687
```

Simpson's rule

```
In [3]: def simpson(f, a, b):
            return (f(a)+4*f((a+b)/2)+f(b))*(b-a)/6

In [4]: simpson(lambda x: x**x, 0.5, 1)

Out[4]: 0.4109013813880978
```

Recall that the degree of accuracy of Simpson's rule is 3. This means the rule integrates polynomials $1, x, x^2, x^3$ exactly, but not x^4. We can use this as a way to verify our code:

```
In [5]: simpson(lambda x: x, 0, 1)

Out[5]: 0.5

In [6]: simpson(lambda x: x**2, 0, 1)

Out[6]: 0.3333333333333333

In [7]: simpson(lambda x: x**3, 0, 1)

Out[7]: 0.25

In [8]: simpson(lambda x: x**4, 0, 1)

Out[8]: 0.20833333333333334
```

Composite Simpson's rule

Next we code the composite Simpson's rule, and verify the result of Example 75.

```
In [9]: def compsimpson(f, a, b, n):
            h = (b-a)/n
            nodes = np.zeros(n+1)
            for i in range(n+1):
                nodes[i] = a+i*h
            sum = f(a)+f(b)
            for i in range(2, n-1, 2):
                sum += 2*f(nodes[i])
            for i in range(1, n, 2):
                sum += 4*f(nodes[i])
            return sum*h/3

In [10]: compsimpson(lambda x: x*np.log(x), 1, 2, 12)

Out[10]: 0.636294560831306
```

Exercise 4.2-3: Determine the value of n required to approximate

$$\int_0^2 \frac{1}{x+1} dx$$

to within 10^{-4}, and compute the approximation, using the composite trapezoidal and composite Simpson's rule.

Composite rules and roundoff error

As we increase n in the composite rules to lower error, the number of function evaluations increases, and a natural question to ask would be whether roundoff error could accumulate and cause problems. Somewhat remarkably, the answer is no. Let's assume the roundoff error associated with computing $f(x)$ is bounded for all x, by some positive constant ϵ. And let's try to compute the roundoff error in composite Simpson rule. Since each function evaluation in the composite rule incorporates an error of (at most) ϵ, the total error is bounded by

$$\frac{h}{3}\left[\epsilon + 2\sum_{j=1}^{\frac{n}{2}-1}\epsilon + 4\sum_{j=1}^{n/2}\epsilon + \epsilon\right] \leq \frac{h}{3}\left[\epsilon + 2\left(\frac{n}{2}-1\right)\epsilon + 4\left(\frac{n}{2}\right)\epsilon + \epsilon\right] = \frac{h}{3}(3n\epsilon) = hn\epsilon.$$

However, since $h = (b-a)/n$, the bound simplifies as $hn\epsilon = (b-a)\epsilon$. Therefore no matter how large n is, that is, how large the number of function evaluations is, the roundoff error is bounded by the same constant $(b-a)\epsilon$ which only depends on the size of the interval.

Exercise 4.2-4: (This problem shows that numerical quadrature is stable with respect to error in function values.) Assume the function values $f(x_i)$ are approximated by $\tilde{f}(x_i)$, so that $|f(x_i) - \tilde{f}(x_i)| < \epsilon$ for any $x_i \in (a, b)$. Find an upper bound on the error of numerical quadrature $\sum w_i f(x_i)$ when it is actually computed as $\sum w_i \tilde{f}(x_i)$.

4.3 Gaussian quadrature

Newton-Cotes formulas were obtained by integrating interpolating polynomials with equally-spaced nodes. The equal spacing is convenient in deriving simple expressions for the composite rules. However, this placement of nodes is not necessarily the optimal placement. For example, the trapezoidal rule approximates the integral by integrating a linear function that joins the endpoints of the function. The fact that this is not the optimal choice can be seen by sketching a simple parabola.

The idea of Gaussian quadrature is the following: in the numerical quadrature rule

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

choose x_i and w_i in such a way that the quadrature rule has the highest possible accuracy. Note that unlike Newton-Cotes formulas where we started labeling the nodes with x_0, in the Gaussian quadrature the first node is x_1. This difference in notation is common in the literature, and each choice makes the subsequent equations in the corresponding theory easier to read.

Example 76. Let $(a, b) = (-1, 1)$, and $n = 2$. Find the "best" x_i and w_i.

There are four parameters to determine: x_1, x_2, w_1, w_2. We need four constraints. Let's require the rule to integrate the following functions exactly: $f(x) = 1, f(x) = x, f(x) = x^2$, and $f(x) = x^3$.

If the rule integrates $f(x) = 1$ exactly, then $\int_{-1}^{1} dx = \sum_{i=1}^{2} w_i$, i.e., $w_1 + w_2 = 2$. If the rule integrates $f(x) = x$ exactly, then $\int_{-1}^{1} x dx = \sum_{i=1}^{2} w_i x_i$, i.e., $w_1 x_1 + w_2 x_2 = 0$. Continuing this for

$f(x) = x^2$, and $f(x) = x^3$, we obtain the following equations:

$$\begin{aligned} w_1 + w_2 &= 2 \\ w_1 x_1 + w_2 x_2 &= 0 \\ w_1 x_1^2 + w_2 x_2^2 &= \frac{2}{3} \\ w_1 x_1^3 + w_2 x_2^3 &= 0. \end{aligned}$$

Solving the equations gives: $w_1 = w_2 = 1, x_1 = \frac{-\sqrt{3}}{3}, x_2 = \frac{\sqrt{3}}{3}$. Therefore the quadrature rule is:

$$\int_{-1}^{1} f(x)dx \approx f\left(\frac{-\sqrt{3}}{3}\right) + f\left(\frac{\sqrt{3}}{3}\right).$$

Observe that:

- The two nodes are not equally spaced on $(-1, 1)$.
- The accuracy of the rule is three, and it uses only two nodes. Recall that the accuracy of Simpson's rule is also three but it uses three nodes. In general Gaussian quadrature gives a degree of accuracy of $2n - 1$ using only n nodes.

We were able to solve for the nodes and weights in the simple example above, however, as the number of nodes increases, the resulting non-linear system of equations will be very difficult to solve. There is an alternative approach using the theory of orthogonal polynomials, a topic we will discuss in detail later. Here, we will use a particular set of orthogonal polynomials $\{L_0(x), L_1(x), ..., L_n(x), ...\}$ called Legendre polynomials. We will give a definition of these polynomials in the next chapter. For this discussion, we just need the following properties of these polynomials:

- $L_n(x)$ is a monic polynomial of degree n for each n.
- $\int_{-1}^{1} P(x)L_n(x)dx = 0$ for any polynomial $P(x)$ of degree less than n.

The first few Legendre polynomials are

$$L_0(x) = 1, L_1(x) = x, L_2(x) = x^2 - \frac{1}{3}, L_3(x) = x^3 - \frac{3}{5}x, L_4(x) = x^4 - \frac{6}{7}x^2 + \frac{3}{35}.$$

How do these polynomials help with finding the nodes and the weights of the Gaussian quadrature rule? The answer is short: the **roots** of the Legendre polynomials are the **nodes** of the quadrature rule!

To summarize, the **Gauss-Legendre quadrature rule** for the integral of f over $(-1, 1)$ is

$$\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w_i f(x_i)$$

where $x_1, x_2, ..., x_n$ are the roots of the nth Legendre polynomial, and the weights are computed using the following theorem.

Theorem 77. *Suppose that $x_1, x_2, ..., x_n$ are the roots of the nth Legendre polynomial $L_n(x)$ and the weights are given by*

$$w_i = \int_{-1}^{1} \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} dx.$$

Then the Gauss-Legendre quadrature rule has degree of accuracy $2n-1$. In other words, if $P(x)$ is any polynomial of degree less than or equal to $2n-1$, then

$$\int_{-1}^{1} P(x)dx = \sum_{i=1}^{n} w_i P(x_i).$$

Proof. Let's start with a polynomial $P(x)$ with degree less than n. Construct the Lagrange interpolant for $P(x)$, using the nodes as $x_1, ..., x_n$

$$P(x) = \sum_{i=1}^{n} P(x_i) l_i(x) = \sum_{i=1}^{n} \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} P(x_i).$$

There is no error term above because the error term depends on the nth derivative of $P(x)$, but $P(x)$ is a polynomial of degree less than n, so that derivative is zero. Integrate both sides to get

$$\begin{aligned}
\int_{-1}^{1} P(x)dx = \int_{-1}^{1} \left[\sum_{i=1}^{n} \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} P(x_i) \right] dx &= \int_{-1}^{1} \left[\sum_{i=1}^{n} P(x_i) \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} \right] dx \\
&= \sum_{i=1}^{n} \left[\int_{-1}^{1} P(x_i) \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} dx \right] \\
&= \sum_{i=1}^{n} \left[\int_{-1}^{1} \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} dx \right] P(x_i) \\
&= \sum_{i=1}^{n} w_i P(x_i).
\end{aligned}$$

Therefore the theorem is correct for polynomials of degree less than n. Now let $P(x)$ be a polynomial of degree greater than or equal to n, but less than or equal to $2n-1$. Divide $P(x)$ by the Legendre polynomial $L_n(x)$ to get

$$P(x) = Q(x)L_n(x) + R(x).$$

Note that

$$P(x_i) = Q(x_i)L_n(x_i) + R(x_i) = R(x_i)$$

since $L_n(x_i) = 0$, for $i = 1, 2, ..., n$. Also observe the following facts:

1. $\int_{-1}^{1} Q(x)L_n(x)dx = 0$, since $Q(x)$ is a polynomial of degree less than n, and from the second property of Legendre polynomials.

2. $\int_{-1}^{1} R(x)dx = \sum_{i=1}^{n} w_i R(x_i)$, since $R(x)$ is a polynomial of degree less than n, and from the first part of the proof.

Putting these facts together we get:

$$\int_{-1}^{1} P(x)dx = \int_{-1}^{1} \left[Q(x)L_n(x) + R(x)\right] dx = \int_{-1}^{1} R(x)dx = \sum_{i=1}^{n} w_i R(x_i) = \sum_{i=1}^{n} w_i P(x_i).$$

□

Table 4.1 displays the roots of the Legendre polynomials L_2, L_3, L_4, L_5 and the corresponding weights.

n	Roots	Weights
2	$\frac{1}{\sqrt{3}} = 0.5773502692$	1
	$-\frac{1}{\sqrt{3}} = -0.5773502692$	1
3	$-(\frac{3}{5})^{1/2} = -0.7745966692$	$\frac{5}{9} = 0.5555555556$
	0.0	$\frac{8}{9} = 0.8888888889$
	$(\frac{3}{5})^{1/2} = 0.7745966692$	$\frac{5}{9} = 0.5555555556$
4	0.8611363116	0.3478548451
	0.3399810436	0.6521451549
	-0.3399810436	0.6521451549
	-0.8611363116	0.3478548451
5	0.9061798459	0.2369268850
	0.5384693101	0.4786286705
	0.0	0.5688888889
	-0.5384693101	0.4786286705
	-0.9061798459	0.2369268850

Table 4.1: Roots of Legendre polynomials L_2 through L_5

Example 78. Approximate $\int_{-1}^{1} \cos x dx$ using Gauss-Legendre quadrature with $n = 3$ nodes.

Solution. From Table 4.1, and using two-digit rounding, we have

$$\int_{-1}^{1} \cos x dx \approx 0.56 \cos(-0.77) + 0.89 \cos 0 + 0.56 \cos(0.77) = 1.69$$

and the true solution is $\sin(1) - \sin(-1) = 1.68$.

So far we discussed integrating functions over the interval $(-1, 1)$. What if we have a different integration domain? The answer is simple: change of variables! To compute $\int_a^b f(x)dx$ for any $a < b$, we use the following change of variables:

$$t = \frac{2x - a - b}{b - a} \Leftrightarrow x = \frac{1}{2}\left[(b - a)t + a + b\right]$$

With this substitution, we have

$$\int_a^b f(x)dx = \frac{b - a}{2} \int_{-1}^{1} f\left(\frac{1}{2}\left[(b - a)t + a + b\right]\right) dt.$$

Now we can approximate the integral on the right-hand side as before.

Example 79. Approximate $\int_{0.5}^{1} x^x dx$ using Gauss-Legendre quadrature with $n = 2$ nodes.

Solution. Transform the integral using $x = \frac{1}{2}(0.5t + 1.5) = \frac{1}{2}(\frac{t}{2} + \frac{3}{2}) = \frac{t+3}{4}, dx = \frac{dt}{4}$ to get:

$$\int_{0.5}^{1} x^x dx = \frac{1}{4} \int_{-1}^{1} \left(\frac{t + 3}{4}\right)^{\frac{t+3}{4}} dt.$$

For $n = 2$

$$\frac{1}{4} \int_{-1}^{1} \left(\frac{t + 3}{4}\right)^{\frac{t+3}{4}} dt \approx \frac{1}{4}\left[\left(\frac{1}{4\sqrt{3}} + \frac{3}{4}\right)^{\left(\frac{1}{4\sqrt{3}} + \frac{3}{4}\right)} + \left(-\frac{1}{4\sqrt{3}} + \frac{3}{4}\right)^{\left(-\frac{1}{4\sqrt{3}} + \frac{3}{4}\right)}\right] = 0.410759,$$

using six significant digits. We will next use Python for a five-node computation of the integral.

Python code for Gauss-Legendre rule with five nodes

The following code computes the Gauss-Legendre rule for $\int_{-1}^{1} f(x)dx$ using $n = 5$ nodes. The nodes and weights are from Table 4.1.

```
In [1]: def gauss(f):
            return 0.2369268851*f(-0.9061798459) + 0.2369268851*f(0.9061798459) + \
                   0.5688888889*f(0) + 0.4786286705*f(0.5384693101) + \
                   0.4786286705*f(-0.5384693101)
```

Now we compute $\frac{1}{4}\int_{-1}^{1}\left(\frac{t+3}{4}\right)^{\frac{t+3}{4}}dt$ using the code:

```
In [2]: 0.25*gauss(lambda t: (t/4+3/4)**(t/4+3/4))

Out[2]: 0.41081564812239885
```

The next theorem is about the error of the Gauss-Legendre rule. Its proof can be found in Atkinson [3]. The theorem shows, in particular, that the degree of accuracy of the quadrature rule, using n nodes, is $2n-1$.

Theorem 80. *Let $f \in C^{2n}[-1,1]$. The error of Gauss-Legendre rule satisfies*

$$\int_a^b f(x)dx - \sum_{i=1}^{n} w_i f(x_i) = \frac{2^{2n+1}(n!)^4}{(2n+1)\left[(2n)!\right]^2}\frac{f^{(2n)}(\xi)}{(2n)!}$$

for some $\xi \in (-1,1)$.

Using Stirling's formula $n! \sim e^{-n}n^n(2\pi n)^{1/2}$, where the symbol $\sim$ means the ratio of the two sides converges to 1 as $n \to \infty$, it can be shown that

$$\frac{2^{2n+1}(n!)^4}{(2n+1)\left[(2n)!\right]^2} \sim \frac{\pi}{4^n}.$$

This means the error of Gauss-Legendre rule decays at an exponential rate of $1/4^n$ as opposed to, for example, the polynomial rate of $1/n^4$ for composite Simpson's rule.

Exercise 4.3-1: Prove that the sum of the weights in Gauss-Legendre quadrature is 2, for any n.

Exercise 4.3-2: Approximate $\int_1^{1.5} x^2 \log x dx$ using Gauss-Legendre rule with $n = 2$ and $n = 3$. Compare the approximations to the exact value of the integral.

Exercise 4.3-3: A composite Gauss-Legendre rule can be obtained similar to composite Newton-Cotes formulas. Consider $\int_0^2 e^x dx$. Divide the interval $(0,2)$ into two subintervals $(0,1),(1,2)$ and apply the Gauss-Legendre rule with two-nodes to each subinterval. Compare the estimate with the estimate obtained from the Gauss-Legendre rule with four-nodes applied to the whole interval $(0,2)$.

4.4 Multiple integrals

The numerical quadrature methods we have discussed can be generalized to higher dimensional integrals. We will consider the two-dimensional integral

$$\int\int_R f(x,y)dA.$$

The domain R determines the difficulty in generalizing the one-dimensional formulas we learned before. The simplest case would be a rectangular domain $R = \{(x,y)|a \leq x \leq b, c \leq y \leq d\}$. We can then write the double integral as the iterated integral

$$\int\int_R f(x,y)dA = \int_a^b \left(\int_c^d f(x,y)dy \right) dx.$$

Consider a numerical quadrature rule

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i).$$

Apply the rule using n_2 nodes to the inner integral to get the approximation

$$\int_a^b \left(\sum_{j=1}^{n_2} w_j f(x,y_j) \right) dx$$

where the y_j's are the nodes. Rewrite, by interchanging the integral and summation, to get

$$\sum_{j=1}^{n_2} w_j \left(\int_a^b f(x,y_j)dx \right)$$

and apply the quadrature rule again, using n_1 nodes, to get the approximation

$$\sum_{j=1}^{n_2} w_j \left(\sum_{i=1}^{n_1} w_i f(x_i,y_j) \right).$$

This gives the two-dimensional rule

$$\int_a^b \left(\int_c^d f(x,y)dy \right) dx \approx \sum_{j=1}^{n_2} \sum_{i=1}^{n_1} w_i w_j f(x_i,y_j).$$

For simplicity, we ignored the error term in the above derivation; however, its inclusion is straightforward.

For an example, let's derive the two-dimensional Gauss-Legendre rule for the integral

$$\int_0^1 \int_0^1 f(x,y)dydx \tag{4.5}$$

using two nodes for each axis. Note that each integral has to be transformed to $(-1,1)$. Start with the inner integral $\int_0^1 f(x,y)dy$ and use

$$t = 2y - 1, dt = 2dy$$

to transform it to

$$\frac{1}{2}\int_{-1}^1 f\left(x, \frac{t+1}{2}\right) dt$$

and apply Gauss-Legendre rule with two nodes to get the approximation

$$\frac{1}{2}\left(f\left(x, \frac{-1/\sqrt{3}+1}{2}\right) + f\left(x, \frac{1/\sqrt{3}+1}{2}\right)\right).$$

Substitute this approximation in (4.5) for the inner integral to get

$$\int_0^1 \frac{1}{2}\left(f\left(x, \frac{-1/\sqrt{3}+1}{2}\right) + f\left(x, \frac{1/\sqrt{3}+1}{2}\right)\right) dx.$$

Now transform this integral to the domain $(-1,1)$ using

$$s = 2x - 1, ds = 2dx$$

to get

$$\frac{1}{4}\int_0^1 \left(f\left(\frac{s+1}{2}, \frac{-1/\sqrt{3}+1}{2}\right) + f\left(\frac{s+1}{2}, \frac{1/\sqrt{3}+1}{2}\right)\right) ds.$$

Apply the Gauss-Legendre rule again to get

$$\begin{aligned}\frac{1}{4}\Bigg[&f\left(\frac{-1/\sqrt{3}+1}{2}, \frac{-1/\sqrt{3}+1}{2}\right) + f\left(\frac{-1/\sqrt{3}+1}{2}, \frac{1/\sqrt{3}+1}{2}\right) \\ &+ f\left(\frac{1/\sqrt{3}+1}{2}, \frac{-1/\sqrt{3}+1}{2}\right) + f\left(\frac{1/\sqrt{3}+1}{2}, \frac{1/\sqrt{3}+1}{2}\right)\Bigg].\end{aligned} \tag{4.6}$$

Figure (4.1) displays the nodes used in this calculation.

Next we derive the two-dimensional Simpson's rule for the same integral, $\int_0^1 \int_0^1 f(x,y)dydx$, using $n = 2$, which corresponds to three nodes in the Simpson's rule (recall that n is the number of nodes in Gauss-Legendre rule, but $n+1$ is the number of nodes in Newton-Cotes formulas).

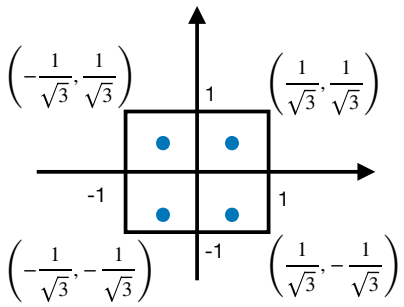


Figure 4.1: Nodes of double Gauss-Legendre rule

The inner integral is approximated as

$$\int_0^1 f(x,y)dy \approx \frac{1}{6}\left(f(x,0) + 4f(x,0.5) + f(x,1)\right).$$

Substitute this approximation for the inner integral in $\int_0^1 \left(\int_0^1 f(x,y)dy\right) dx$ to get

$$\frac{1}{6}\int_0^1 \left(f(x,0) + 4f(x,0.5) + f(x,1)\right) dx.$$

Apply Simpson's rule again to this integral with $n = 2$ to obtain the final approximation:

$$\frac{1}{6}\left[\frac{1}{6}\Big(f(0,0) + 4f(0,0.5) + f(0,1) + 4(f(0.5,0) + 4f(0.5,0.5) + f(0.5,1)) + f(1,0) + 4f(1,0.5) + f(1,1)\Big)\right]. \tag{4.7}$$

Figure (4.2) displays the nodes used in the above calculation.

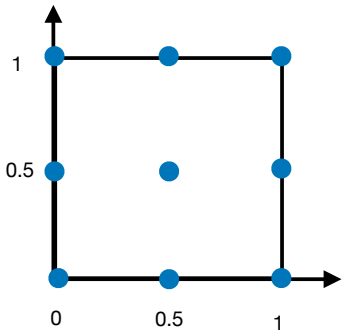


Figure 4.2: Nodes of double Simpson's rule

For a specific example, consider the integral

$$\int_0^1 \int_0^1 \left(\frac{\pi}{2}\sin \pi x\right)\left(\frac{\pi}{2}\sin \pi y\right) dydx.$$

This integral can be evaluated exactly, and its value is 1. It is used as a test integral for numerical quadrature rules. Evaluating equations (4.6) and (4.7) with $f(x, y) = \left(\frac{\pi}{2}\sin \pi x\right)\left(\frac{\pi}{2}\sin \pi y\right)$, we obtain the approximations given in the table below:

Simpson's rule (9 nodes)	Gauss-Legendre (4 nodes)	Exact integral
1.0966	0.93685	1

The Gauss-Legendre rule gives a slightly better estimate than Simpson's rule, but using less than half the number of nodes.

The approach we have discussed can be extended to regions that are not rectangular, and to higher dimensions. More details, including algorithms for double and triple integrals using Simpson's and Gauss-Legendre rule, can be found in Burden, Faires, Burden [4].

There is however, an obvious disadvantage of the way we have generalized one-dimensional quadrature rules to higher dimensions. Imagine a numerical integration problem where the dimension is 360; such high dimensions appear in some problems from financial engineering. Even if we used two nodes for each dimension, the total number of nodes would be 2^{360}, which is about 10^{100}, a number too large. In very large dimensions, the only general method for numerical integration is the Monte Carlo method. In Monte Carlo, we generate pseudorandom numbers from the domain, and evaluate the function average at those points. In other words,

$$\int_R f(\boldsymbol{x})d\boldsymbol{x} \approx \frac{1}{n}\sum_{i=1}^{n} f(\boldsymbol{x}_i)$$

where $\boldsymbol{x}_i$ are pseudorandom vectors uniformly distributed in R. For the two-dimensional integral we discussed before, the Monte Carlo estimate is

$$\int_a^b \int_c^d f(x, y)dydx \approx \frac{(b-a)(d-c)}{n}\sum_{i=1}^{n} f(a+(b-a)x_i, c+(d-c)y_i) \tag{4.8}$$

where x_i, y_i are pseudorandom numbers from $(0, 1)$. In Python, the function **np.random.rand()** generates a pseudorandom number from the uniform distribution between 0 and 1. The following code takes the endpoints of the intervals a, b, c, d, and the number of nodes n, which is called the sample size in the Monte Carlo literature, and returns the estimate for the integral using Equation (4.8).

```
In [1]: import numpy as np

In [2]: def mc(f, a, b, c, d, n):
```

```
    sum = 0.
    for i in range(n):
        sum += f(a+(b-a)*np.random.rand(),
                 c+(d-c)*np.random.rand())*(b-a)*(d-c)
    return sum/n
```

Now we use Monte Carlo to estimate the integral $\int_0^1 \int_0^1 \left(\frac{\pi}{2}\sin \pi x\right)\left(\frac{\pi}{2}\sin \pi y\right) dydx$:

```
In [2]: mc((x,y)->(pi^2/4)*sin(pi*x)*sin(pi*y),0,1,0,1,500)
```

```
Out[2]: 0.9441778334708931
```

With $n = 500$, we obtain a Monte Carlo estimate of 0.944178. An advantage of the Monte Carlo method is its simplicity: the above code can be easily generalized to higher dimensions. A disadvantage of Monte Carlo is its slow rate of convergence, which is $O(1/\sqrt{n})$. Figure (4.3) displays 500 pseudorandom vectors from the unit square.

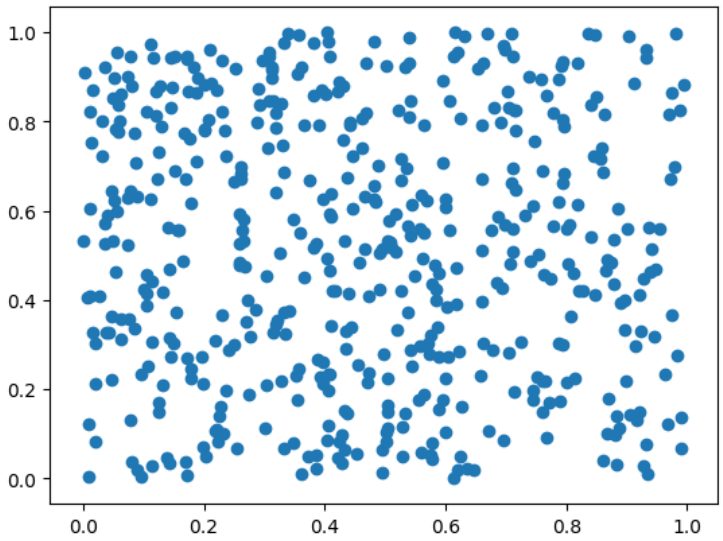


Figure 4.3: Monte Carlo: 500 pseudorandom vectors

Example 81. Capstick & Keister [5] discuss some high dimensional test integrals, some with analytical solutions, motivated by physical applications such as the calculation of quantum mechanical matrix elements in atomic, nuclear, and particle physics. One of the integrals with a known solution is

$$\int_{\mathbb{R}^s} \cos(\|\boldsymbol{t}\|) e^{-\|\boldsymbol{t}\|^2} dt_1 dt_2 \cdots dt_s$$

where $\|\boldsymbol{t}\| = (t_1^2 + \ldots + t_s^2)^{1/2}$. This integral can be transformed to an integral over the s-dimensional

unit cube as

$$\pi^{s/2} \int_{(0,1)^s} \cos\left[\left(\frac{(F^{-1}(x_1))^2 + \ldots + (F^{-1}(x_s))^2}{2}\right)^{1/2}\right] dx_1 dx_2 \cdots dx_s \tag{4.9}$$

where F^{-1} is the inverse of the cumulative distribution function of the standard normal distribution:

$$F(x) = \frac{1}{(2\pi)^{1/2}} \int_{-\infty}^{x} e^{-s^2/2} ds.$$

We will estimate the integral (4.9) by Monte Carlo as

$$\frac{\pi^{s/2}}{n} \sum_{i=1}^{n} \cos\left[\left(\frac{(F^{-1}(x_1^{(i)}))^2 + \ldots + (F^{-1}(x_s^{(i)}))^2}{2}\right)^{1/2}\right]$$

where $\boldsymbol{x}^{(i)} = (x_1^{(i)}, \ldots, x_s^{(i)})$ is an s-dimensional vector of uniform random numbers between 0 and 1.

```
In [1]: import numpy as np
```

```
In [2]: import matplotlib.pyplot as plt
        %matplotlib inline
```

The following algorithm, known as the Beasley-Springer-Moro algorithm [8], gives an approximation to $F^{-1}(x)$.

```
In [3]: def invNormal(u):
            # Beasley-Springer-Moro algorithm
            a0 = 2.50662823884
            a1 = -18.61500062529
            a2 = 41.39119773534
            a3 = -25.44106049637
            b0 = -8.47351093090
            b1 = 23.08336743743
            b2 = -21.06224101826
            b3 = 3.13082909833
            c0 = 0.3374754822726147
            c1 = 0.9761690190917186
            c2 = 0.1607979714918209
            c3 = 0.0276438810333863
            c4 = 0.0038405729373609
            c5 = 0.0003951896511919
```

```
        c6 = 0.0000321767881768
        c7 = 0.0000002888167364
        c8 = 0.0000003960315187

        y = u-0.5
        if np.abs(y)<0.42:
            r = y*y
            x = y*(((a3*r+a2)*r+a1)*r+a0)/((((b3*r+b2)*r+b1)*r+b0)*r+1)
        else:
            r = u
            if y>0:
                r = 1-u
            r = np.log(-np.log(r))
            x = c0+r*(c1+r*(c2+r*(c3+r*(c4+r*(c5+r*(c6+r*(c7+r*c8)))))))
            if y<0:
                x = -x
        return x
```

The following is the Monte Carlo estimate of the integral. It takes the dimension s and the sample size n as inputs.

```
In [4]: def mc(s, n):
            est = 0
            for j in range(n):
                sum = 0
                for i in range(s):
                    sum += (invNormal(np.random.rand()))**2
                est += np.cos((sum/2)**0.5)
            return np.pi**(s/2)*est/n
```

The exact value of the integral for $s = 25$ is 1.356914×10^6. The following code computes the relative error of the Monte Carlo estimate with sample size n.

```
In [5]: relerror = lambda n: np.abs(mc(25,n)+1.356914*10**6)/(1.356914*10**6)
```

Let's plot the relative error of some Monte Carlo estimates. First, we generate sample sizes from 50,000 to 1,000,000 in increments of 50,000.

```
In [6]: samples = [n for n in range(50000,1000001, 50000)]
```

For each sample size, we compute the relative error, and then plot the results.

```
In [7]: error = [relerror(n) for n in samples]

In [8]: plt.plot(samples, error)
        plt.xlabel('Sample size (n)')
        plt.ylabel('Relative error');
```

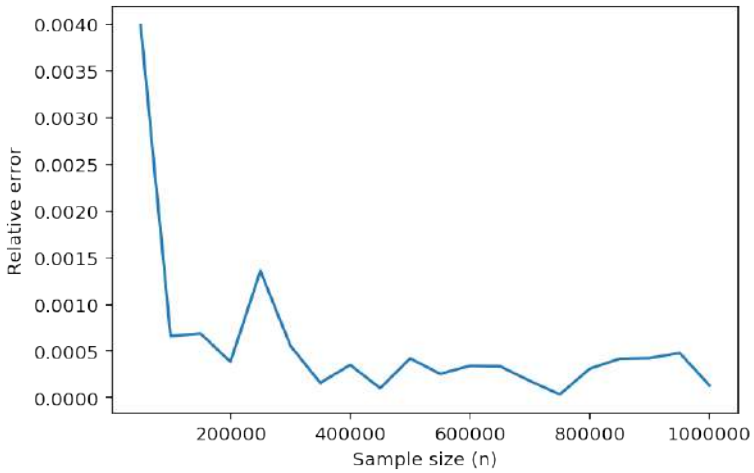


Figure 4.4: Monte Carlo relative error for the integral (4.9)

4.5 Improper integrals

The quadrature rules we have learned so far cannot be applied (or applied with a poor performance) to integrals such as $\int_a^b f(x)dx$ if $a, b = \pm\infty$ or if a, b are finite but f is not continuous at one or both of the endpoints: recall that both Newton-Cotes and Gauss-Legendre error bound theorems require the integrand to have a number of continuous derivatives on the closed interval $[a, b]$. For example, an integral in the form

$$\int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}} dx$$

clearly cannot be approximated using the trapezoidal or Simpson's rule without any modifications, since both rules require the values of the integrand at the end points which do not exist. One could try using the Gauss-Legendre rule, but the fact that the integrand does not satisfy the smoothness conditions required by the Gauss-Legendre error bound means the error of the approximation might be large.

A simple remedy to the problem of improper integrals is to change the variable of integration and transform the integral, if possible, to one that behaves well.

Example 82. Consider the previous integral $\int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}} dx$. Try the transformation $\theta = \cos^{-1} x$.

Then $d\theta = -dx/\sqrt{1-x^2}$ and

$$\int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}} dx = -\int_{\pi}^{0} f(\cos\theta) d\theta = \int_{0}^{\pi} f(\cos\theta) d\theta.$$

The latter integral can be evaluated using, for example, Simpson's rule, provided f is smooth on $[0, \pi]$.

If the interval of integration is infinite, another approach that might work is truncation of the interval to a finite one. The success of this approach depends on whether we can estimate the resulting error.

Example 83. Consider the improper integral $\int_0^\infty e^{-x^2} dx$. Write the integral as

$$\int_0^\infty e^{-x^2} dx = \int_0^t e^{-x^2} dx + \int_t^\infty e^{-x^2} dx,$$

where t is the "level of truncation" to determine. We can estimate the first integral on the right-hand side using a quadrature rule. The second integral on the right-hand side is the error due to approximating $\int_0^\infty e^{-x^2} dx$ by $\int_0^t e^{-x^2} dx$. An upper bound for the error can be found easily for this example: note that when $x \geq t$, $x^2 = xx \geq tx$, thus

$$\int_t^\infty e^{-x^2} dx \leq \int_t^\infty e^{-tx} dx = e^{-t^2}/t.$$

When $t = 5$, $e^{-t^2}/t \approx 10^{-12}$, therefore, approximating the integral $\int_0^\infty e^{-x^2} dx$ by $\int_0^5 e^{-x^2} dx$ will be accurate within 10^{-12}. Additional error will come from estimating the latter integral by numerical quadrature.

4.6 Numerical differentiation

The derivative of f at x_0 is

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

This formula gives an obvious way to estimate the derivative by

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

for small h. What this formula lacks, however, is it does not give any information about the error of the approximation.

We will try another approach. Similar to Newton-Cotes quadrature, we will construct the interpolating polynomial for f, and then use the derivative of the polynomial as an approximation for the derivative of f.

Let's assume $f \in C^2(a,b), x_0 \in (a,b)$, and $x_0 + h \in (a,b)$. Construct the linear Lagrange interpolating polynomial $p_1(x)$ for the data $(x_0, f(x_0)), (x_1, f(x_1)) = (x_0 + h, f(x_0 + h))$. From Theorem 51, we have

$$\begin{aligned} f(x) &= \underbrace{\frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1)}_{p_1(x)} + \underbrace{\frac{f''(\xi(x))}{2!}(x - x_0)(x - x_1)}_{\text{interpolation error}} \\ &= \frac{x - (x_0 + h)}{x_0 - (x_0 + h)} f(x_0) + \frac{x - x_0}{x_0 + h - x_0} f(x_0 + h) + \frac{f''(\xi(x))}{2!}(x - x_0)(x - x_0 - h) \\ &= \frac{x - x_0 - h}{-h} f(x_0) + \frac{x - x_0}{h} f(x_0 + h) + \frac{f''(\xi(x))}{2!}(x - x_0)(x - x_0 - h). \end{aligned}$$

Now let's differentiate $f(x)$:

$$\begin{aligned} f'(x) = -\frac{f(x_0)}{h} + \frac{f(x_0 + h)}{h} &+ f''(\xi(x)) \frac{d}{dx}\left[\frac{(x - x_0)(x - x_0 - h)}{2}\right] \\ &+ \frac{(x - x_0)(x - x_0 - h)}{2} \frac{d}{dx}\left[f''(\xi(x))\right] \\ = \frac{f(x_0 + h) - f(x_0)}{h} + \frac{2x - 2x_0 - h}{2} f''(\xi(x)) &+ \frac{(x - x_0)(x - x_0 - h)}{2} f'''(\xi(x))\xi'(x). \end{aligned}$$

In the above equation, we know ξ is between x_0 and $x_0 + h$; however, we have no knowledge about $\xi'(x)$, which appears in the last term. Fortunately, if we set $x = x_0$, the term with $\xi'(x)$ vanishes and we get:

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \frac{h}{2} f''(\xi(x_0)).$$

This formula is called the **forward-difference** formula if $h > 0$ and **backward-difference** formula if $h < 0$. Note that from this formula we can obtain a bound on the error

$$\left| f'(x_0) - \frac{f(x_0 + h) - f(x_0)}{h} \right| \leq \frac{h}{2} \sup_{x \in (x_0, x_0 + h)} f''(x).$$

To obtain the forward-difference and backward-difference formulas we started with a linear polynomial interpolant on two points. Using more points and a higher order interpolant gives more accuracy, but also increases the computing time and roundoff error. In general, let $f \in C^{n+1}[a,b]$ and $x_0, x_1, ..., x_n$ are distinct numbers in $[a,b]$. We have

$$\begin{aligned} f(x) &= \sum_{k=0}^{n} f(x_k) l_k(x) + f^{(n+1)}(\xi) \frac{(x - x_0)(x - x_1) \cdots (x - x_n)}{(n+1)!} \\ \Rightarrow f'(x) &= \sum_{k=0}^{n} f(x_k) l_k'(x) + f^{(n+1)}(\xi) \frac{d}{dx}\left[\frac{(x - x_0)(x - x_1) \cdots (x - x_n)}{(n+1)!}\right] \\ &\quad + \frac{(x - x_0)(x - x_1) \cdots (x - x_n)}{(n+1)!} \frac{d}{dx}(f^{(n+1)}(\xi)). \end{aligned}$$

If $x = x_j$ for $j = 0, 1, ..., n$, the last term vanishes, and using the following result

$$\frac{d}{dx}\left[(x-x_0)(x-x_1)\cdots(x-x_n)\right]_{x=x_j} = \prod_{k=0,k\neq j}^{n}(x_j - x_k)$$

we obtain

$$f'(x_j) = \sum_{k=0}^{n} f(x_k)l'_k(x_j) + \frac{f^{(n+1)}(\xi(x_j))}{(n+1)!}\prod_{k=0,k\neq j}^{n}(x_j - x_k) \tag{4.10}$$

which is called the $(n+1)$-point formula to approximate $f'(x_j)$. The most common formulas use $n = 2$ and $n = 4$. Here we discuss $n = 2$, that is, three-point formulas. The nodes are, x_0, x_1, x_2. The Lagrange basis polynomials and their derivatives are:

$$\begin{aligned}
l_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} \Rightarrow l'_0(x) = \frac{2x-x_1-x_2}{(x_0-x_1)(x_0-x_2)} \\
l_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \Rightarrow l'_1(x) = \frac{2x-x_0-x_2}{(x_1-x_0)(x_1-x_2)} \\
l_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \Rightarrow l'_2(x) = \frac{2x-x_0-x_1}{(x_2-x_0)(x_2-x_1)}
\end{aligned}$$

These derivatives can be substituted in (4.10) to obtain the three-point formula. We can simplify these formulas if the nodes are spaced equally, that is, $x_1 = x_0 + h, x_2 = x_1 + h = x_0 + 2h$. Then, we obtain

$$f'(x_0) = \frac{1}{2h}\left[-3f(x_0) + 4f(x_0+h) - f(x_0+2h)\right] + \frac{h^2}{3}f^{(3)}(\xi_0) \tag{4.11}$$

$$f'(x_0+h) = \frac{1}{2h}\left[-f(x_0) + f(x_0+2h)\right] - \frac{h^2}{6}f^{(3)}(\xi_1) \tag{4.12}$$

$$f'(x_0+2h) = \frac{1}{2h}\left[f(x_0) - 4f(x_0+h) + 3f(x_0+2h)\right] + \frac{h^2}{3}f^{(3)}(\xi_2). \tag{4.13}$$

It turns out that the first and third equations ((4.11) and (4.13)) are equivalent. To see this, first substitute x_0 by $x_0 - 2h$ in the third equation to get (ignoring the error term)

$$f'(x_0) = \frac{1}{2h}\left[f(x_0-2h) - 4f(x_0-h) + 3f(x_0)\right],$$

and then set h to $-h$ in the right-hand side to get $\frac{1}{2h}\left[-f(x_0+2h) + 4f(x_0+h) - 3f(x_0)\right]$, which gives us the first equation.

Therefore we have only two distinct equations, (4.11) and (4.12). We rewrite these equations below with one modification: in (4.12), we substitute x_0 by $x_0 - h$. We then obtain two different

formulas for $f'(x_0)$:

$$f'(x_0) = \frac{-3f(x_0) + 4f(x_0+h) - f(x_0+2h)}{2h} + \frac{h^2}{3}f^{(3)}(\xi_0) \rightarrow \text{three-point endpoint formula}$$
$$f'(x_0) = \frac{f(x_0+h) - f(x_0-h)}{2h} - \frac{h^2}{6}f^{(3)}(\xi_1) \rightarrow \text{three-point midpoint formula}$$

The three-point midpoint formula has some advantages: it has half the error of the endpoint formula, and it has one less function evaluation. The endpoint formula is useful if one does not know the value of f on one side, a situation that may happen if x_0 is close to an endpoint.

Example 84. The following table gives the values of $f(x) = \sin x$. Estimate $f'(0.1), f'(0.3)$ using an appropriate three-point formula.

x	$f(x)$
0.1	0.09983
0.2	0.19867
0.3	0.29552
0.4	0.38942

Solution. To estimate $f'(0.1)$, we set $x_0 = 0.1$, and $h = 0.1$. Note that we can only use the three-point endpoint formula.

$$f'(0.1) \approx \frac{1}{0.2}(-3(0.09983) + 4(0.19867) - 0.29552) = 0.99835.$$

The correct answer is $\cos 0.1 = 0.995004$.

To estimate $f'(0.3)$ we can use the midpoint formula:

$$f'(0.3) \approx \frac{1}{0.2}(0.38942 - 0.19867) = 0.95375.$$

The correct answer is $\cos 0.3 = 0.955336$ and thus the absolute error is 1.59×10^{-3}. If we use the endpoint formula to estimate $f'(0.3)$ we set $h = -0.1$ and compute

$$f'(0.3) \approx \frac{1}{-0.2}(-3(0.29552) + 4(0.19867) - 0.09983) = 0.95855$$

with an absolute error of 3.2×10^{-3}.

Exercise 4.6-1: In some applications, we want to estimate the derivative of an unknown function from empirical data. However, empirical data usually come with "noise", that is, error due to data collection, data reporting, or some other reason. In this exercise we will investigate how stable the difference formulas are when there is noise in the data. Consider the following data obtained from $y = e^x$. The data is exact to six digits. We estimate $f'(1.01)$ using the

x	1.00	1.01	1.02
$f(x)$	2.71828	2.74560	2.77319

three-point midpoint formula and obtain $f'(1.01) = \frac{2.77319-2.71828}{0.02} = 2.7455$. The true value is $f'(1.01) = e^{1.01} = 2.74560$, and the relative error due to rounding is 3.6×10^{-5}.

Next, we add some noise to the data: we increase 2.77319 by 10% to 3.050509, and decrease 2.71828 by 10% to 2.446452. Here is the noisy data:

x	1.00	1.01	1.02
$f(x)$	**2.446452**	2.74560	**3.050509**

Estimate $f'(1.01)$ using the noisy data, and compute its relative error. How does the relative error compare with the relative error for the non-noisy data?

We next want to explore how to estimate the second derivative of f. A similar approach to estimating f' can be taken and the second derivative of the interpolating polynomial can be used as an approximation. Here we will discuss another approach, using Taylor expansions. Expand f about x_0, and evaluate it at $x_0 + h$ and $x_0 - h$:

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f^{(3)}(x_0) + \frac{h^4}{24}f^{(4)}(\xi_+)$$
$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) - \frac{h^3}{6}f^{(3)}(x_0) + \frac{h^4}{24}f^{(4)}(\xi_-)$$

where ξ_+ is between x_0 and $x_0 + h$, and ξ_- is between x_0 and $x_0 - h$. Add the equations to get

$$f(x_0 + h) + f(x_0 - h) = 2f(x_0) + h^2 f''(x_0) + \frac{h^4}{24}\left[f^{(4)}(\xi_+) + f^{(4)}(\xi_-)\right].$$

Solving for $f''(x_0)$ gives

$$f''(x_0) = \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} - \frac{h^2}{24}\left[f^{(4)}(\xi_+) + f^{(4)}(\xi_-)\right].$$

Note that $\frac{f^{(4)}(\xi_+)+f^{(4)}(\xi_-)}{2}$ is a number between $f^{(4)}(\xi_+)$ and $f^{(4)}(\xi_-)$, so from the Intermediate Value Theorem 6, we can conclude there exists some ξ between ξ_- and ξ_+ so that

$$f^{(4)}(\xi) = \frac{f^{(4)}(\xi_+) + f^{(4)}(\xi_-)}{2}.$$

Then the above formula simplifies as

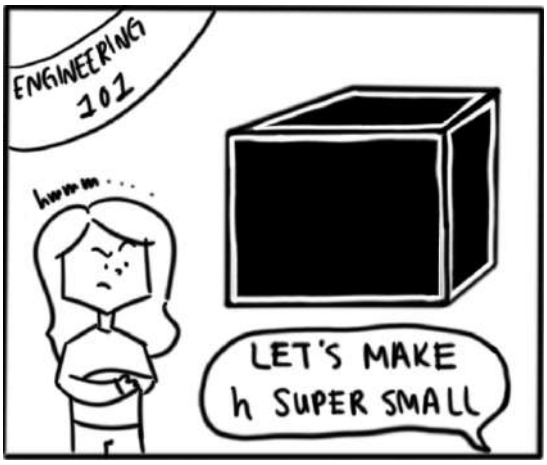
$$f''(x_0) = \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} - \frac{h^2}{12}f^{(4)}(\xi)$$

for some ξ between $x_0 - h$ and $x_0 + h$.

Numerical differentiation and roundoff error

Arya and the mysterious black box

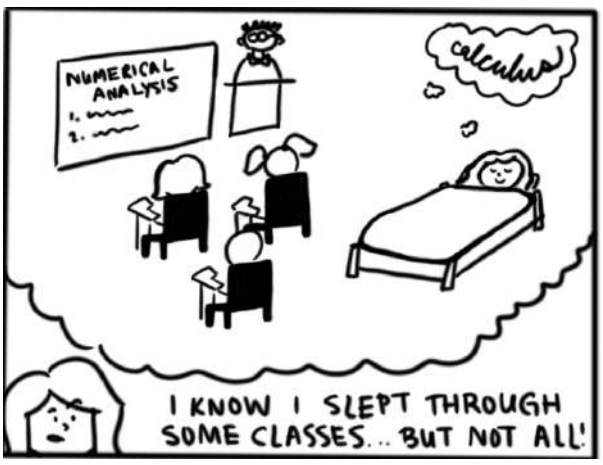
College life is full of mysteries, and Arya faces one in an engineering class: a black box! What is a black box? It is a computer program, or some device, which produces an output when an input is provided. We do not know the inner workings of the system, and hence comes the name black box. Let's think of the black box as a function f, and represent the input and output as $x, f(x)$. Of course, we do not have a formula for f.

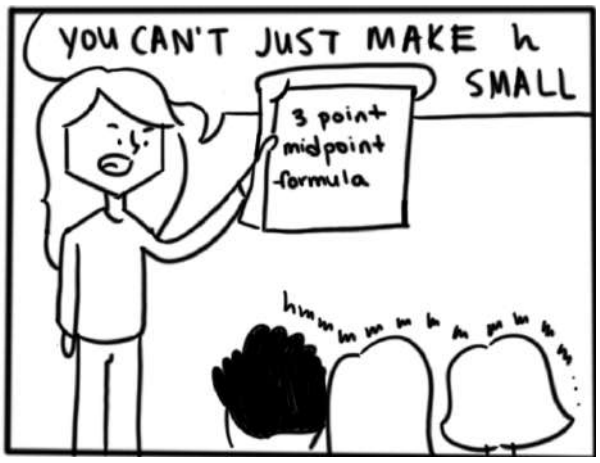


What Arya's engineering classmates want to do is compute the derivative information of the black box, that is, $f'(x)$, when $x = 2$. (The input to this black box can be any real number.) Students want to use the three-point midpoint formula to estimate $f'(2)$:

$$f'(2) \approx \frac{1}{2h}\left[f(2+h) - f(2-h)\right].$$

They argue how to pick h in this formula. One of them says they should make h as small as possible, like 10^{-8}. Arya is skeptical. She mutters to herself, "I know I slept through some of my numerical analysis lectures, but not all!"





She tells her classmates about the cancellation of leading digits phenomenon, and to make her point more convincing, she makes the following experiment: let $f(x) = e^x$, and suppose we want

to compute $f'(2)$ which is e^2. Arya uses the three-point midpoint formula above to estimate $f'(2)$, for various values of h, and for each case she computes the absolute value of the difference between the three-point midpoint estimate and the exact solution e^2. She tabulates her results in the table below. Clearly, smaller h does not result in smaller error. In this experiment, $h = 10^{-6}$ gives the smallest error.

h	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}
Abs. error	1.2×10^{-8}	1.9×10^{-10}	5.2×10^{-11}	7.5×10^{-9}	2.1×10^{-8}

Theoretical analysis

Numerical differentiation is a numerically unstable problem. To reduce the truncation error, we need to decrease h, which in turn increases the roundoff error due to cancellation of significant digits in the function difference calculation. Let $e(x)$ denote the roundoff error in computing $f(x)$ so that $f(x) = \tilde{f}(x) + e(x)$, where $\tilde{f}$ is the value computed by the computer. Consider the three-point midpoint formula:

$$\begin{aligned}
&\left| f'(x_0) - \frac{\tilde{f}(x_0+h) - \tilde{f}(x_0-h)}{2h} \right| \\
&= \left| f'(x_0) - \frac{f(x_0+h) - e(x_0+h) - f(x_0-h) + e(x_0-h)}{2h} \right| \\
&= \left| f'(x_0) - \frac{f(x_0+h) - f(x_0-h)}{2h} + \frac{e(x_0-h) - e(x_0+h)}{2h} \right| \\
&= \left| -\frac{h^2}{6} f^{(3)}(\xi) + \frac{e(x_0-h) - e(x_0+h)}{2h} \right| \leq \frac{h^2}{6} M + \frac{\epsilon}{h}
\end{aligned}$$

where we assumed $|f^{(3)}(\xi)| \leq M$ and $|e(x)| \leq \epsilon$. To reduce the truncation error $h^2 M/6$ one would decrease h, which would then result in an increase in the roundoff error ϵ/h. An optimal value for h can be found with these assumptions using Calculus: find the value for h that minimizes the function $s(h) = \frac{Mh^2}{6} + \frac{\epsilon}{h}$. The answer is $h = \sqrt[3]{3\epsilon/M}$.

Let's revisit the table Arya presented where 10^{-6} was found to be the optimal value for h. The calculations were done using Python, which reports 15 digits when asked for e^2. Let's assume all these digits are correct, and thus let $\epsilon = 10^{-16}$. Since $f^{(3)}(x) = e^x$ and $e^2 \approx 7.4$, let's take $M = 7.4$. Then

$$h = \sqrt[3]{3\epsilon/M} = \sqrt[3]{3 \times 10^{-16}/7.4} \approx 3.4 \times 10^{-6}$$

which is in good agreement with the optimal value 10^{-6} of Arya's numerical results.

Exercise 4.6-2: Find the optimal value for h that will minimize the error for the formula

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \frac{h}{2}f''(\xi)$$

in the presence of roundoff error, using the approach of Section 4.6.

a) Consider estimating $f'(1)$ where $f(x) = x^2$ using the above formula. What is the optimal value for h for estimating $f'(1)$, assuming that the roundoff error is bounded by $\epsilon = 10^{-16}$ (which is the machine epsilon 2^{-53} in the 64-bit floating point representation).

b) Use Python to compute

$$f_n'(1) = \frac{f(1+10^{-n}) - f(1)}{10^{-n}},$$

for $n = 1, 2, ..., 20$, and describe what happens.

c) Discuss your findings in parts (a) and (b) and how they relate to each other.

Exercise 4.6-3: The function $\int_0^x \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$ is related to the distribution function of the standard normal random variable, a very important distribution in probability and statistics. Often times we want to solve equations like

$$\int_0^x \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt = z \tag{4.14}$$

for x, where z is some real number between 0 and 1. This can be done by using Newton's method to solve the equation $f(x) = 0$ where

$$f(x) = \int_0^x \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt - z.$$

Note that from Fundamental Theorem of Calculus, $f'(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$. Newton's method will require the calculation of

$$\int_0^{p_k} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt \tag{4.15}$$

where p_k is a Newton iterate. This integral can be computed using numerical quadrature. Write a Python code that takes z as its input, and outputs x, such that Equation (4.14) holds. In your code, use the Python codes for Newton's method and the composite Simpson's rule you were given in class. For Newton's method set tolerance to 10^{-5} and $p_0 = 0.5$, and for composite Simpson's rule take $n = 10$, when computing the integrals (4.15) that appear in Newton iteration. Then run your code for $z = 0.4$ and $z = 0.1$, and report your output.

Chapter 5

Approximation Theory

5.1 Discrete least squares

Arya's adventures in the physics lab

College life is expensive, and Arya is happy to land a job working at a physics lab for some extra cash. She does some experiments, some data analysis, and a little grading. In one experiment she conducted, where there is one independent variable x, and one dependent variable y, she was asked to plot y against x values. (There are a total of six data points.) She gets the following plot:

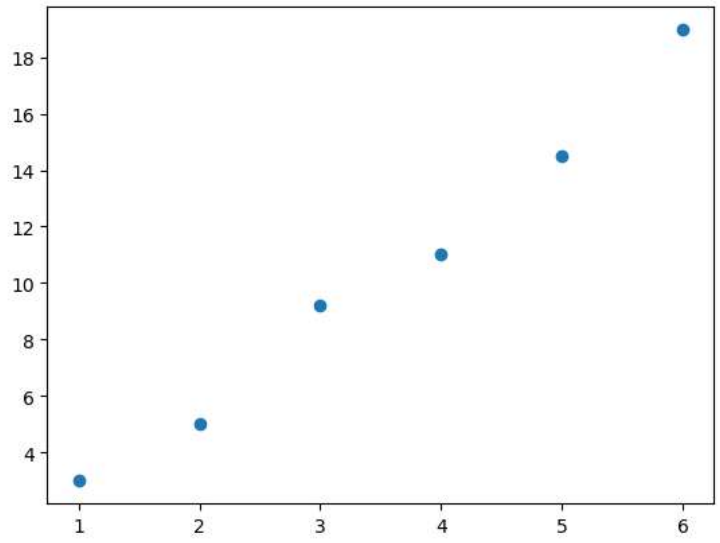


Figure 5.1: Scatter plot of data

Arya's professor thinks the relationship between the variables should be linear, but we do not see data falling on a perfect line because of measurement error. The professor is not happy, professors are usually not happy when lab results act up, and asks Arya to come up with a linear formula,

something like $y = ax + b$, to explain the relationship. Arya first thinks about interpolation, but quickly realizes it is not a good idea. (Why?). Let's help Arya with her problem.

Analysis of the problem

Let's try passing a line through the data points. Figure (5.2) plots one such line, $y = 3x - 0.5$, together with the data points.

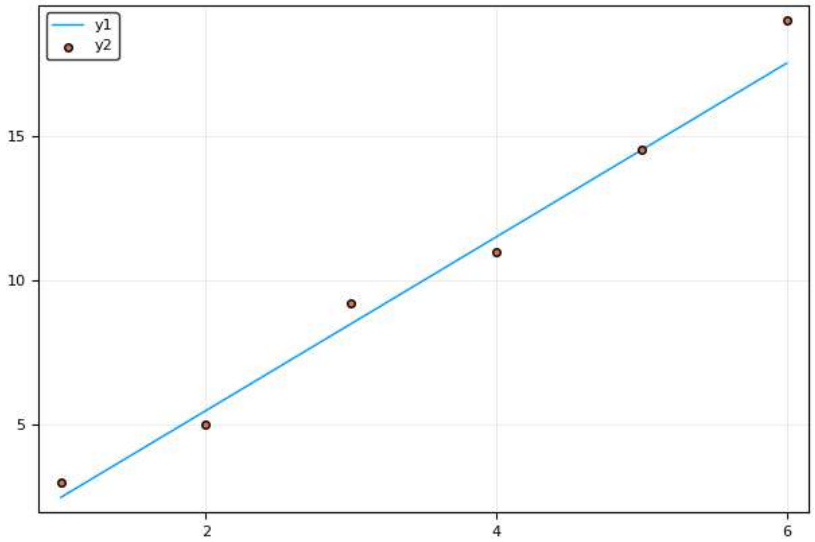


Figure 5.2: Data with an approximating line

There are certainly many other choices we have for the line: we could increase or decrease the slope a little, change the intercept a bit, and obtain multiple lines that have a visually good fit to the data. The crucial question is, how can we decide which line is the "best" line, among all the possible lines? If we can quantify how good the fit of a given line is to the data, and come up with a notion for error, perhaps then we can find the line that minimizes this error.

Let's generalize the problem a little. We have:

- Data: $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$

and we want to find a line that gives the "best" approximation to the data:

- Linear approximation: $y = f(x) = ax + b$

The questions we want to answer are:

1. What does "best" approximation mean?
2. How do we find a, b that gives the line with the "best" approximation?

Observe that for each x_i, there is the corresponding y_i of the data point, and $f(x_i) = ax_i + b$, which is the predicted value by the linear approximation. We can measure error by considering the deviations between the actual y coordinates and the predicted values:

$$(y_1 - ax_1 - b), (y_2 - ax_2 - b), ..., (y_m - ax_m - b)$$

There are several ways we can form a measure of error using these deviations, and each approach gives a different line approximating the data. The best approximation means finding a, b that minimizes the error measured in one of the following ways:

- $E = \max_i \{|y_i - ax_i - b|\}$; minimax problem
- $E = \sum_{i=1}^m |y_i - ax_i - b|$; absolute deviations
- $E = \sum_{i=1}^m (y_i - ax_i - b)^2$; least squares problem

In this chapter we will discuss the least squares problem, the simplest one among the three options. We want to minimize

$$E = \sum_{i=1}^{m} (y_i - ax_i - b)^2$$

with respect to the parameters a, b. For a minimum to occur, we must have

$$\frac{\partial E}{\partial a} = 0 \text{ and } \frac{\partial E}{\partial b} = 0.$$

We have:

$$\frac{\partial E}{\partial a} = \sum_{i=1}^{m} \frac{\partial E}{\partial a}(y_i - ax_i - b)^2 = \sum_{i=1}^{m} (-2x_i)(y_i - ax_i - b) = 0$$
$$\frac{\partial E}{\partial b} = \sum_{i=1}^{m} \frac{\partial E}{\partial b}(y_i - ax_i - b)^2 = \sum_{i=1}^{m} (-2)(y_i - ax_i - b) = 0$$

Using algebra, these equations can be simplified as

$$b\sum_{i=1}^{m} x_i + a\sum_{i=1}^{m} x_i^2 = \sum_{i=1}^{m} x_i y_i$$
$$bm + a\sum_{i=1}^{m} x_i = \sum_{i=1}^{m} y_i,$$

which are called the **normal equations**. The solution to this system of equations is

$$a = \frac{m\sum_{i=1}^m x_i y_i - \sum_{i=1}^m x_i \sum_{i=1}^m y_i}{m\left(\sum_{i=1}^m x_i^2\right) - \left(\sum_{i=1}^m x_i\right)^2}, b = \frac{\sum_{i=1}^m x_i^2 \sum_{i=1}^m y_i - \sum_{i=1}^m x_i y_i \sum_{i=1}^m x_i}{m\left(\sum_{i=1}^m x_i^2\right) - \left(\sum_{i=1}^m x_i\right)^2}.$$

Let's consider a slightly more general question. Given data

- Data: $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$

can we find the best polynomial approximation

- Polynomial approximation: $P_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$

where m will be usually much larger than n. Similar to the above discussion, we want to minimize

$$E = \sum_{i=1}^{m} (y_i - P_n(x_i))^2 = \sum_{i=1}^{m} \left(y_i - \sum_{j=0}^{n} a_j x_i^j \right)^2$$

with respect to the parameters $a_n, a_{n-1}, ..., a_0$. For a minimum to occur, the necessary conditions are

$$\frac{\partial E}{\partial a_k} = 0 \Rightarrow -\sum_{i=1}^{m} y_i x_i^k + \sum_{j=0}^{n} a_j \left(\sum_{i=1}^{m} x_i^{k+j} \right) = 0$$

for $k = 0, 1, ..., n$. (we are skipping some algebra here!) The **normal equations** for polynomial approximation are

$$\sum_{j=0}^{n} a_j \left(\sum_{i=1}^{m} x_i^{k+j} \right) = \sum_{i=1}^{m} y_i x_i^k \tag{5.1}$$

for $k = 0, 1, ..., n$. This is a system of $(n+1)$ equations and $(n+1)$ unknowns. We can write this system as a matrix equation

$$A\boldsymbol{a} = \boldsymbol{b} \tag{5.2}$$

where $\boldsymbol{a}$ is the unknown vector we are trying to find, and $\boldsymbol{b}$ is the constant vector

$$\boldsymbol{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} y_i x_i \\ \vdots \\ \sum_{i=1}^{m} y_i x_i^n \end{bmatrix}$$

and A is an $(n+1)$ by $(n+1)$ symmetric matrix with (k, j)th entry $A_{kj}, k = 1, ..., n+1, j = 1, 2, ..., n+1$ given by

$$A_{kj} = \sum_{i=1}^{m} x_i^{k+j-2}.$$

The equation $A\boldsymbol{a} = \boldsymbol{b}$ has a unique solution if the x_i are distinct, and $n \leq m-1$. Solving this equation by computing the inverse matrix A^{-1} is not advisable, since there could be significant roundoff error. Next, we will write a Python code for least squares approximation, and use the Python function *np.linalg.solve*(A, b) to solve the matrix equation $A\boldsymbol{a} = \boldsymbol{b}$ for $\boldsymbol{a}$. The *np.linalg.solve* function in

Python uses numerically optimized matrix factorizations based on the LAPACK routine to solve the matrix equation. More details on this topic can be found in Heath [10] (Chapter 3).

Python code for least squares approximation

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

The function **leastsqfit** takes the x- and y-coordinates of the data, and the degree of the polynomial we want to use, n, as inputs. It solves the matrix Equation (5.2).

```
In [2]: def leastsqfit(x, y, n):
            m = x.size # number of data points
            d = n+1 # number of coefficients to be determined
            A = np.zeros((d, d))
            b = np.zeros(d)
            # the linear system we want to solve is Ax=b
            p = np.zeros(2*n+1)
            for k in range(d):
                sum = 0
                for i in range(m):
                    sum += y[i]*x[i]**k
                b[k] = sum
            # p[i] below is the sum of the i-th power of the x coordinates
            p[0] = m
            for i in range(1, 2*n+1):
                sum = 0
                for j in range(m):
                    sum += x[j]**i
                p[i] = sum
            # We next compute the upper triangular part of the coefficient
            # matrix A, and its diagonal
            for k in range(d):
                for j in range(k, d):
                    A[k, j] = p[k+j]
            # The lower triangular part of the matrix is defined using the
            # fact the matrix is symmetric
            for i in range(1, d):
                for j in range(i):
```

```
                A[i, j] = A[j, i]
        a = np.linalg.solve(A, b)
        return a
```

Here is the data used to produce the first plot of the chapter: Arya's data:

```
In [3]: xd = np.array([1, 2, 3, 4, 5, 6])
        yd = np.array([3, 5, 9.2, 11, 14.5, 19])
```

We fit a least squares line to the data:

```
In [4]: leastsqfit(xd, yd, 1)
```

```
Out[4]: array([-0.74666667,  3.15142857])
```

The polynomial is $-0.746667 + 3.15143x$. The next function **poly(x,a)** takes the output of $a =$ **leastsqfit**, and evaluates the least squares polynomial at x.

```
In [5]: def poly(x, a):
            d = a.size
            sum = 0
            for i in range(d):
                sum += a[i]*x**i
            return sum
```

For example, if we want to compute the least squares line at 3.5, we call the following functions:

```
In [6]: a = leastsqfit(xd, yd, 1)
        poly(3.5, a)
```

```
Out[6]: 10.283333333333335
```

The next function computes the least squares error: $E = \sum_{i=1}^{m}(y_i - p_n(x_i))^2$. It takes the output of $a =$ **leastsqfit**, and the data, as inputs.

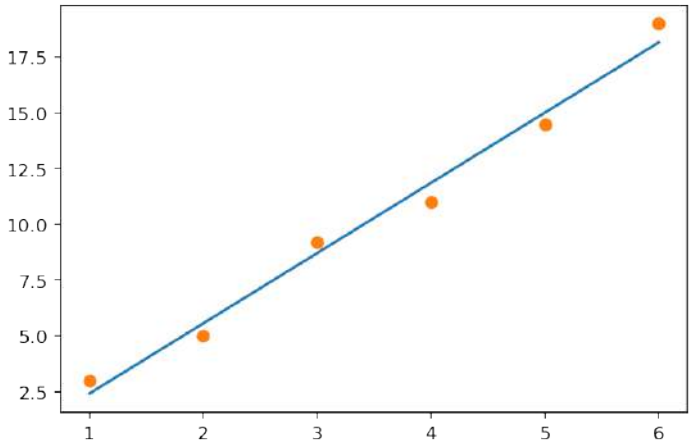
```
In [7]: def leastsqerror(a, x, y):
            sum = 0
            m = y.size
            for i in range(m):
                sum += (y[i]-poly(x[i],a))**2
            return sum
```

```
In [8]: a = leastsqfit(xd, yd, 1)
        leastsqerror(a, xd, yd)
```

```
Out[8]: 2.6070476190476177
```

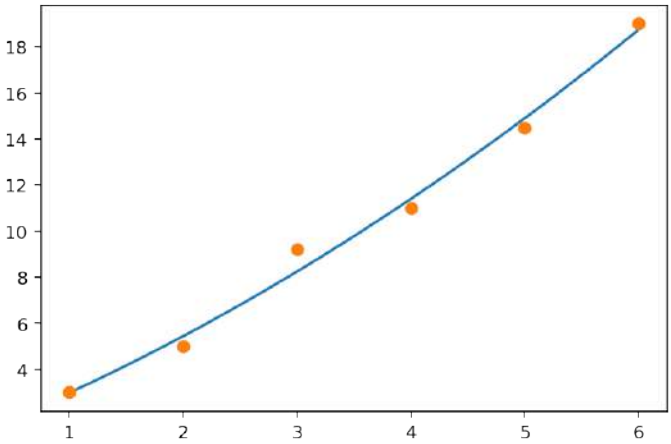
Next we plot the least squares line and the data together.

```
In [9]: a = leastsqfit(xd, yd, 1)
        xaxis = np.linspace(1, 6, 500)
        yvals = poly(xaxis, a)
        plt.plot(xaxis, yvals)
        plt.plot(xd, yd, 'o');
```



We try a second degree polynomial in least squares approximation next.

```
In [10]: a = leastsqfit(xd, yd, 2)
         xaxis = np.linspace(1, 6, 500)
         yvals = poly(xaxis, a)
         plt.plot(xaxis, yvals)
         plt.plot(xd, yd, 'o');
```



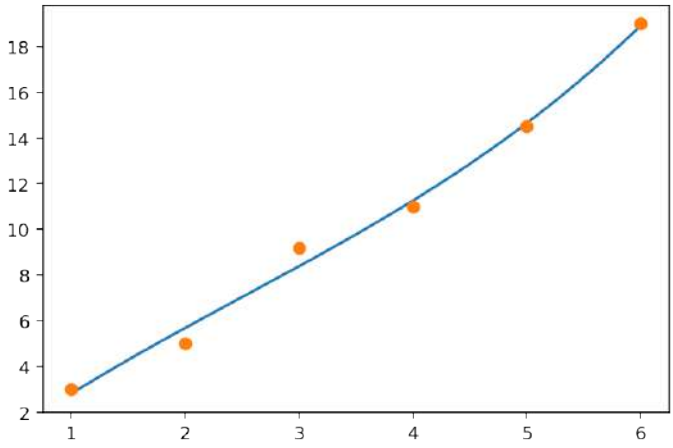
The corresponding error is:

```
In [11]: leastsqerror(a, xd, yd)
```

```
Out[11]: 1.4869285714285714
```

The next polynomial is of degree three:

```
In [12]: a = leastsqfit(xd, yd, 3)
         xaxis = np.linspace(1, 6, 500)
         yvals = poly(xaxis, a)
         plt.plot(xaxis, yvals)
         plt.plot(xd, yd, 'o');
```



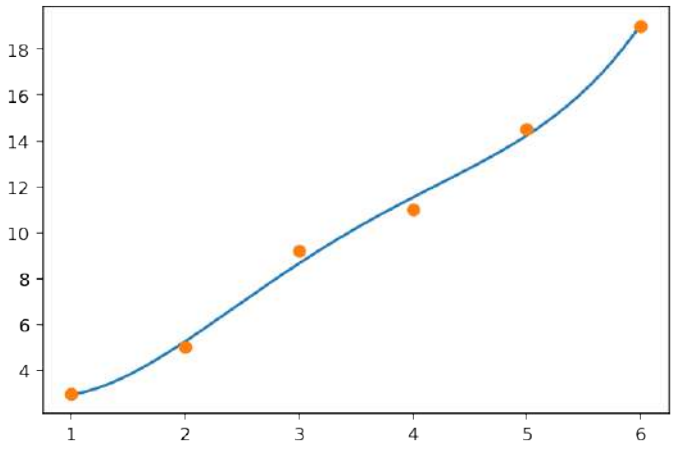
The corresponding error is:

```
In [13]: leastsqerror(a, xd, yd)
```

```
Out[13]: 1.2664285714285732
```

The next polynomial is of degree four:

```
In [14]: a = leastsqfit(xd, yd, 4)
         xaxis = np.linspace(1, 6, 500)
         yvals = poly(xaxis, a)
         plt.plot(xaxis, yvals)
         plt.plot(xd, yd, 'o');
```



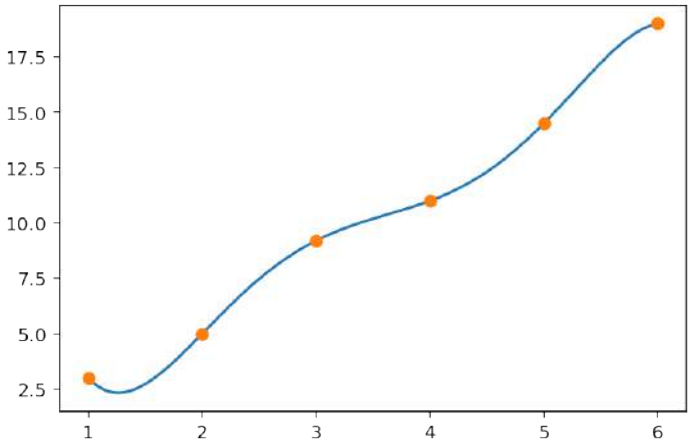
The least squares error is:

```
In [15]: leastsqerror(a, xd, yd)
```

```
Out[15]: 0.723214285714292
```

Finally, we try a fifth degree polynomial. Recall that the normal equations have a unique solution when x_i are distinct, and $n \leq m - 1$. Since $m = 6$ in this example, $n = 5$ is the largest degree with guaranteed unique solution.

```
In [16]: a = leastsqfit(xd, yd, 5)
         xaxis = np.linspace(1, 6, 500)
         yvals = poly(xaxis, a)
         plt.plot(xaxis, yvals)
         plt.plot(xd, yd, 'o');
```



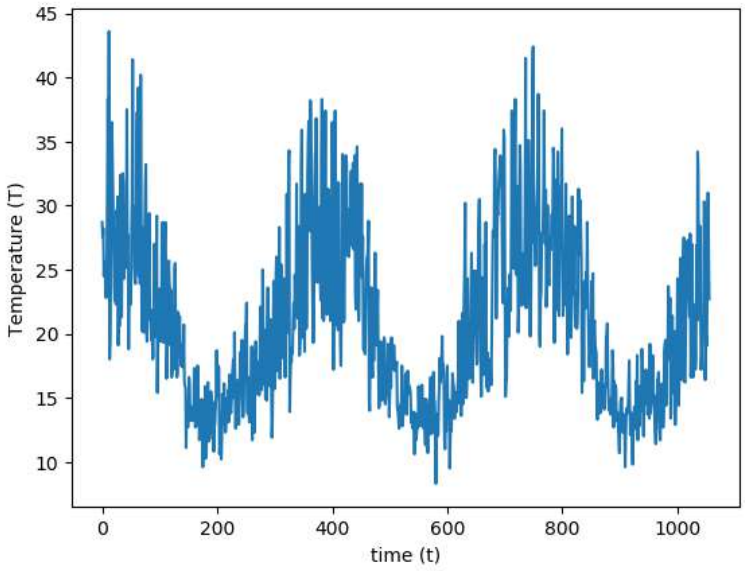
The approximating polynomial of degree five is the interpolating polynomial! What is the least squares error?

Least squares with non-polynomials

The method of least squares is not only for polynomials. For example, suppose we want to find the function

$$f(t) = a + bt + c\sin(2\pi t/365) + d\cos(2\pi t/365) \tag{5.3}$$

that has the best fit to some data $(t_1, T_1), ..., (t_m, T_m)$ in the least-squares sense. This function is used in modeling weather temperature data, where t denotes time, and T denotes the temperature. The following figure plots the daily maximum temperature during a period of 1,056 days, from 2016 until November 21, 2018, as measured by a weather station at Melbourne airport, Australia[1].



To find the best fit function of the form (5.3), we write the least squares error term

$$E = \sum_{i=1}^{m}(f(t_i) - T_i)^2 = \sum_{i=1}^{m}\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right)^2,$$

and set its partial derivatives with respect to the unknowns a, b, c, d to zero to obtain the normal

[1] http://www.bom.gov.au/climate/data/

equations:

$$\begin{aligned}\frac{\partial E}{\partial a} = 0 \Rightarrow \sum_{i=1}^{m} 2\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right) = 0 \\ \Rightarrow \sum_{i=1}^{m}\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right) = 0, \end{aligned} \tag{5.4}$$

$$\begin{aligned}\frac{\partial E}{\partial b} = 0 \Rightarrow \sum_{i=1}^{m}(2t_i)\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right) = 0 \\ \Rightarrow \sum_{i=1}^{m} t_i\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right) = 0, \end{aligned} \tag{5.5}$$

$$\begin{aligned}\frac{\partial E}{\partial c} = 0 \Rightarrow \sum_{i=1}^{m}\left(2\sin\left(\frac{2\pi t_i}{365}\right)\right)\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right) = 0 \\ \Rightarrow \sum_{i=1}^{m}\sin\left(\frac{2\pi t_i}{365}\right)\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right) = 0, \end{aligned} \tag{5.6}$$

$$\begin{aligned}\frac{\partial E}{\partial d} = 0 \Rightarrow \sum_{i=1}^{m}\left(2\cos\left(\frac{2\pi t_i}{365}\right)\right)\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right) = 0 \\ \Rightarrow \sum_{i=1}^{m}\cos\left(\frac{2\pi t_i}{365}\right)\left(a + bt_i + c\sin\left(\frac{2\pi t_i}{365}\right) + d\cos\left(\frac{2\pi t_i}{365}\right) - T_i\right) = 0. \end{aligned} \tag{5.7}$$

Rearranging terms in equations (5.4, 5.5, 5.6, 5.7), we get a system of four equations and four

unknowns:

$$
\begin{aligned}
&am + b\sum_{i=1}^{m} t_i + c\sum_{i=1}^{m} \sin\left(\frac{2\pi t_i}{365}\right) + d\sum_{i=1}^{m} \cos\left(\frac{2\pi t_i}{365}\right) = \sum_{i=1}^{m} T_i \\
&a\sum_{i=1}^{m} t_i + b\sum_{i=1}^{m} t_i^2 + c\sum_{i=1}^{m} t_i \sin\left(\frac{2\pi t_i}{365}\right) + d\sum_{i=1}^{m} t_i \cos\left(\frac{2\pi t_i}{365}\right) = \sum_{i=1}^{m} T_i t_i \\
&a\sum_{i=1}^{m} \sin\left(\frac{2\pi t_i}{365}\right) + b\sum_{i=1}^{m} t_i \sin\left(\frac{2\pi t_i}{365}\right) + c\sum_{i=1}^{m} \sin^2\left(\frac{2\pi t_i}{365}\right) + d\sum_{i=1}^{m} \sin\left(\frac{2\pi t_i}{365}\right)\cos\left(\frac{2\pi t_i}{365}\right) \\
&\qquad = \sum_{i=1}^{m} T_i \sin\left(\frac{2\pi t_i}{365}\right) \\
&a\sum_{i=1}^{m} \cos\left(\frac{2\pi t_i}{365}\right) + b\sum_{i=1}^{m} t_i \cos\left(\frac{2\pi t_i}{365}\right) + c\sum_{i=1}^{m} \sin\left(\frac{2\pi t_i}{365}\right)\cos\left(\frac{2\pi t_i}{365}\right) + d\sum_{i=1}^{m} \cos^2\left(\frac{2\pi t_i}{365}\right) \\
&\qquad = \sum_{i=1}^{m} T_i \cos\left(\frac{2\pi t_i}{365}\right)
\end{aligned}
$$

Using a short-hand notation where we suppress the argument $\left(\frac{2\pi t_i}{365}\right)$ in the trigonometric functions, and the summation indices, we write the above equations as a matrix equation:

$$
\underbrace{\begin{bmatrix} m & \sum t_i & \sum \sin(\cdot) & \sum \cos(\cdot) \\ \sum t_i & \sum t_i^2 & \sum t_i \sin(\cdot) & \sum t_i \cos(\cdot) \\ \sum \sin(\cdot) & \sum t_i \sin(\cdot) & \sum \sin^2(\cdot) & \sum \sin(\cdot)\cos(\cdot) \\ \sum \cos(\cdot) & \sum t_i \cos(\cdot) & \sum \sin(\cdot)\cos(\cdot) & \sum \cos^2(\cdot) \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \underbrace{\begin{bmatrix} \sum T_i \\ \sum T_i t_i \\ \sum T_i \sin(\cdot) \\ \sum T_i \cos(\cdot) \end{bmatrix}}_{\mathbf{r}}
$$

Next, we will use Python to load the data and define the matrices $A, \mathbf{r}$, and then solve the equation $A\mathbf{x} = \mathbf{r}$, where $\mathbf{x} = [a, b, c, d]^T$.

We will use a package called pandas to import data. We import it in our notebook, along with NumPy and Matplotlib:

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

We assume that the data which consists of temperatures is downloaded as a csv file in the same directory where the Python notebook is stored. Make sure the data has no missing entries. The function **pd.read_csv** imports the data into Python as a table (dataframe):

```
In [2]: df = pd.read_csv('WeatherData.csv')
        df
```

```
Out[2]:       Temp
        0     28.7
        1     27.5
        2     28.2
        3     24.5
        4     25.6
        ...    ...
        1051  19.1
        1052  27.1
        1053  31.0
        1054  27.0
        1055  22.7

        [1056 rows x 1 columns]
```

The next step is to store the part of the data we need as an array. In our table there is only one column named **Temp**.

```
In [3]: temp = df['Temp'].to_numpy()
        temp

Out[3]: array([28.7, 27.5, 28.2, ..., 31. , 27. , 22.7])
```

Let's check the type of **temp**, its first entry, and its length:

```
In [4]: type(temp)

Out[4]: numpy.ndarray

In [5]: temp[0]

Out[5]: 28.7

In [6]: temp.size

Out[6]: 1056
```

There are 1,056 temperature values. The x-coordinates are the days, numbered $t = 1, 2, ..., 1056$. Here is the array that stores these time values:

```
In [7]: time = np.arange(1, 1057)
```

Next we define the matrix A, taking advantage of the fact that the matrix is symmetric. The function **np.sum(x)** adds the entries of the array **x**.

```
In [8]: A = np.zeros((4,4))
        A[0,0] = 1056
        A[0,1] = np.sum(time)
        A[0,2] = np.sum(np.sin(2*np.pi*time/365))
        A[0,3] = np.sum(np.cos(2*np.pi*time/365))
        A[1,1] = np.sum(time**2)
        A[1,2] = np.sum(time*np.sin(2*np.pi*time/365))
        A[1,3] = np.sum(time*np.cos(2*np.pi*time/365))
        A[2,2] = np.sum(np.sin(2*np.pi*time/365)**2)
        A[2,3] = np.sum(np.sin(2*np.pi*time/365)*np.cos(2*np.pi*time/365))
        A[3,3] = np.sum(np.cos(2*np.pi*time/365)**2)
        for i in range(1,4):
            for j in range(i):
                A[i,j] = A[j,i]

In [9]: A

Out[9]: array([[ 1.05600000e+03,  5.58096000e+05,  1.22956884e+01,
                -3.62432800e+01],
               [ 5.58096000e+05,  3.93085616e+08, -5.04581481e+04,
                -3.84772834e+04],
               [ 1.22956884e+01, -5.04581481e+04,  5.42338826e+02,
                 1.09944004e+01],
               [-3.62432800e+01, -3.84772834e+04,  1.09944004e+01,
                 5.13661174e+02]])
```

Now we define the vector **r**. The function **np.dot(x,y)** takes the dot product of the arrays x, y. For example, **np.dot**$([1,2,3],[4,5,6]) = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$.

```
In [10]: r = np.zeros((4,1))
         r[0] = np.sum(temp)
         r[1] = np.dot(temp, time)
         r[2] = np.dot(temp, np.sin(2*np.pi*time/365))
         r[3] = np.dot(temp, np.cos(2*np.pi*time/365))

In [11]: r

Out[11]: array([[2.18615000e+04],
                [1.13803102e+07],
                [1.74207707e+03],
                [2.78776127e+03]])
```

We can solve the matrix equation now.

```
In [12]: np.linalg.solve(A, r)
```

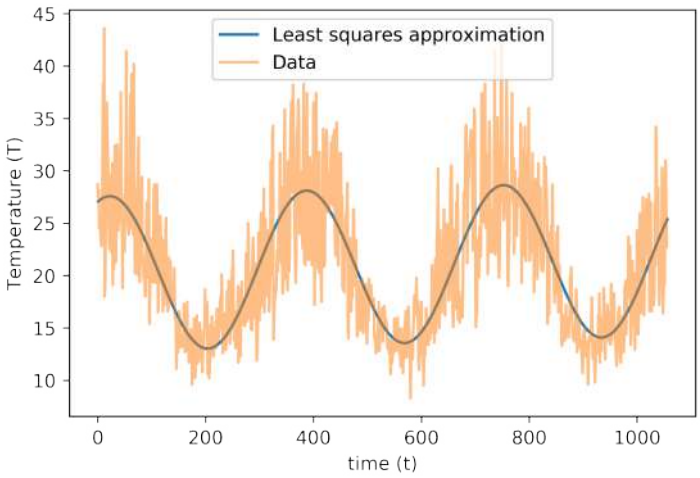
```
Out[12]: array([[2.02897563e+01],
                [1.16773021e-03],
                [2.72116176e+00],
                [6.88808561e+00]])
```

Recall that these constants are the values of a, b, c, d in the definition of $f(t)$. Here is the best fitting function to the data:

```
In [13]: f = lambda t: 20.1393 + 0.00144928*t + 2.72174*np.sin(2*np.pi*t/365) + \
         6.88460*np.cos(2*np.pi*t/365)
```

We next plot the data together with $f(t)$:

```
In [14]: xaxis = np.arange(1, 1057)
         yvals = f(xaxis)
         plt.plot(xaxis, yvals, label='Least squares approximation')
         plt.xlabel('time (t)')
         plt.ylabel('Temperature (T)')
         plt.plot(temp, linestyle='-', alpha=0.5, label='Data')
         plt.legend(loc='upper center');
```



Linearizing data

For another example of non-polynomial least squares, consider finding the function $f(x) = be^{ax}$ with the best least squares fit to some data $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$. We need to find a, b that minimize

$$E = \sum_{i=1}^{m} (y_i - be^{ax_i})^2.$$

The normal equations are

$$\frac{\partial E}{\partial a} = 0 \text{ and } \frac{\partial E}{\partial b} = 0,$$

however, unlike the previous example, this is not a system of linear equations in the unknowns a, b. In general, a root finding type method is needed to solve these equations.

There is a simpler approach we can use when we suspect the data is exponentially related. Consider again the function we want to fit:

$$y = be^{ax}. \tag{5.8}$$

Take the logarithm of both sides:

$$\log y = \log b + ax$$

and rename the variables as $Y = \log y, B = \log b$. Then we obtain the expression

$$Y = ax + B \tag{5.9}$$

which is a linear equation in the transformed variable. In other words, if the original variable y is related to x via Equation (5.8), then $Y = \log y$ is related to x via a linear relationship given by Equation (5.9). So, the new approach is to fit the least squares line $Y = ax + B$ to the data

$$(x_1, \log y_1), (x_2, \log y_2), ..., (x_m, \log y_m).$$

However, it is important to realize that the least squares fit to the transformed data is not necessarily the same as the least squares fit to the original data. The reason is the deviations which least squares minimize are distorted in a non-linear way by the transformation.

Example 85. Consider the following data

x	0	1	2	3	4	5
y	3	5	8	12	23	37

to which we will fit $y = be^{ax}$ in the least-squares sense. The following table displays the data $(x_i, \log y_i)$, using two-digits:

x	0	1	2	3	4	5
$Y = \log y$	1.1	1.6	2.1	2.5	3.1	3.6

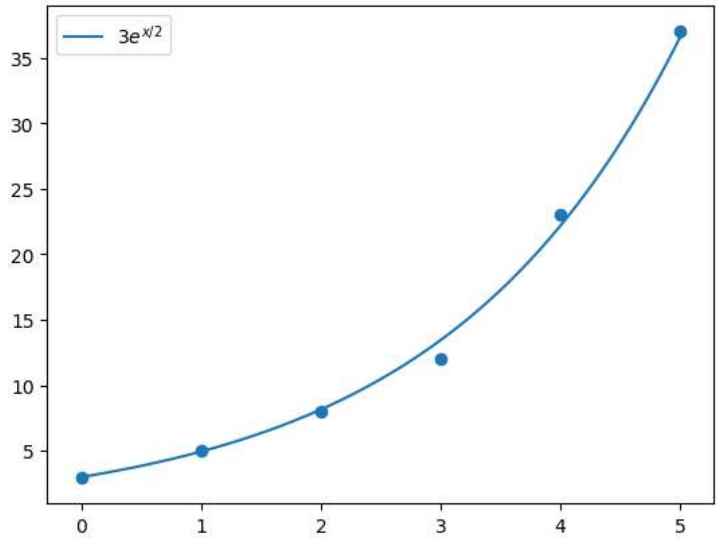
We use the Python code **leastsqfit** to fit a line to this data:

```
In [1]: x = np.array([0, 1, 2, 3, 4, 5])
        y = np.array([1.1, 1.6, 2.1, 2.5, 3.1, 3.6])

In [2]: leastsqfit(x, y, 1)

Out[2]: array([1.09047619, 0.49714286])
```

Therefore the least squares line, using two-digits, is

$$Y = 0.5x + 1.1.$$

This equation corresponds to Equation (5.9), with $a = 0.5$ and $B = 1.1$. We want to obtain the corresponding exponential Equation (5.8), where $b = e^B$. Since $e^{1.1} = 3$, the best fitting exponential function to the data is $y = 3e^{x/2}$. The following graph plots $y = 3e^{x/2}$ together with the data.



Exercise 5.1-1: Find the function of the form $y = ae^x + b\sin(4x)$ that best fits the data below in the least squares sense.

x	1	2	3	4	5
y	-4	6	-1	5	20

Plot the function and the data together.

Exercise 5.1-2: Power-law type relationships are observed in many empirical data. Two variables y, x are said to be related via a power-law if $y = kx^{\alpha}$, where k, α are some constants. The following data[2] lists the top 10 family names in the order of occurrence according to Census 2000. Investigate whether relative frequency of occurrences and the rank of the name are related via a power-law, by

a) Let y be the relative frequencies (number of occurrences divided by the total number of occurrences), and x be the rank, that is, 1 through 10.

b) Use least squares to find a function of the form $y = kx^{\alpha}$. Use linearization.

c) Plot the data together with the best fitting function found in part (b).

Name	Number of Occurrences
Smith	2,376,206
Johnson	1,857,160
Williams	1,534,042
Brown	1,380,145
Jones	1,362,755
Miller	1,127,803
Davis	1,072,335
Garcia	858,289
Rodriguez	804,240
Wilson	783,051

5.2 Continuous least squares

In discrete least squares, our starting point was a set of data points. Here we will start with a continuous function f on $[a, b]$ and answer the following question: how can we find the "best" polynomial $P_n(x) = \sum_{j=0}^{n} a_j x^j$ of degree at most n that approximates f on $[a, b]$? As before, "best" polynomial will mean the polynomial that minimizes the least squares error:

$$E = \int_a^b \left(f(x) - \sum_{j=0}^{n} a_j x^j \right)^2 dx. \tag{5.10}$$

Compare this expression with that of the **discrete least squares**:

$$E = \sum_{i=1}^{m} \left(y_i - \sum_{j=0}^{n} a_j x_i^j \right)^2.$$

[2] https://www.census.gov/topics/population/genealogy/data/2000_surnames.html

To minimize E in (5.10) we set $\frac{\partial E}{\partial a_k} = 0$, for $k = 0, 1, ..., n$, and observe

$$\begin{aligned}\frac{\partial E}{\partial a_k} &= \frac{\partial}{\partial a_k}\left(\int_a^b f^2(x)dx - 2\int_a^b f(x)\left(\sum_{j=0}^n a_j x^j\right)dx + \int_a^b \left(\sum_{j=0}^n a_j x^j\right)^2 dx\right) \\ &= -2\int_a^b f(x)x^k dx + 2\sum_{j=0}^n a_j \int_a^b x^{j+k}dx = 0,\end{aligned}$$

which gives the $(n+1)$ normal equations for the **continuous least squares problem**:

$$\sum_{j=0}^n a_j \int_a^b x^{j+k}dx = \int_a^b f(x)x^k dx \tag{5.11}$$

for $k = 0, 1, ..., n$. Note that the only unknowns in these equations are the a_j's; hence this is a linear system of equations. It is instructive to compare these normal equations with those of the **discrete least squares problem**:

$$\sum_{j=0}^n a_j \left(\sum_{i=1}^m x_i^{k+j}\right) = \sum_{i=1}^m y_i x_i^k.$$

Example 86. Find the least squares polynomial approximation of degree 2 to $f(x) = e^x$ on $(0, 2)$.

Solution. The normal equations are:

$$\sum_{j=0}^2 a_j \int_0^2 x^{j+k}dx = \int_0^2 e^x x^k dx$$

$k = 0, 1, 2$. Here are the three equations:

$$\begin{aligned} a_0\int_0^2 dx + a_1\int_0^2 xdx + a_2\int_0^2 x^2dx &= \int_0^2 e^x dx \\ a_0\int_0^2 xdx + a_1\int_0^2 x^2dx + a_2\int_0^2 x^3dx &= \int_0^2 e^x xdx \\ a_0\int_0^2 x^2dx + a_1\int_0^2 x^3dx + a_2\int_0^2 x^4dx &= \int_0^2 e^x x^2dx \end{aligned}$$

Computing the integrals we get

$$\begin{aligned} 2a_0 + 2a_1 + \frac{8}{3}a_2 &= e^2 - 1 \\ 2a_0 + \frac{8}{3}a_1 + 4a_2 &= e^2 + 1 \\ \frac{8}{3}a_0 + 4a_1 + \frac{32}{5}a_2 &= 2e^2 - 2 \end{aligned}$$

whose solution is $a_0 = 3(-7+e^2)$, $a_1 = -\frac{3}{2}(-37+5e^2)$, $a_2 = \frac{15}{4}(-7+e^2)$. Then

$$P_2(x) = 1.17 + 0.08x + 1.46x^2.$$

The solution method we have discussed for the least squares problem by solving the normal equations as a matrix equation has certain drawbacks:

- The integrals $\int_a^b x^{i+j}dx = \left(b^{i+j+1} - a^{i+j+1}\right)/(i+j+1)$ in the coefficient matrix give rise to matrix equation that is prone to roundoff error.
- There is no easy way to go from $P_n(x)$ to $P_{n+1}(x)$ (which we might want to do if we were not satisfied by the approximation provided by P_n).

There is a better approach to solve the discrete and continuous least squares problem using the orthogonal polynomials we encountered in Gaussian quadrature. Both discrete and continuous least squares problem tries to find a polynomial $P_n(x) = \sum_{j=0}^n a_j x^j$ that satisfies some properties. Notice how the polynomial is written in terms of the monomial basis functions x^j and recall how these basis functions caused numerical difficulties in interpolation. That was the reason we discussed different basis functions like Lagrange and Newton for the interpolation problem. So the idea is to write $P_n(x)$ in terms of some other basis functions

$$P_n(x) = \sum_{j=0}^n a_j \phi_j(x)$$

which would then update the normal equations for continuous least squares (5.11) as

$$\sum_{j=0}^n a_j \int_a^b \phi_j(x)\phi_k(x)dx = \int_a^b f(x)\phi_k(x)dx$$

for $k = 0, 1, ..., n$. The normal equations for the discrete least squares (5.1) gets a similar update:

$$\sum_{j=0}^n a_j \left(\sum_{i=1}^m \phi_j(x_i)\phi_k(x_i)\right) = \sum_{i=1}^m y_i\phi_k(x_i).$$

Going forward, the crucial observation is that the integral of the product of two functions $\int \phi_j(x)\phi_k(x)dx$, or the summation of the product of two functions evaluated at some discrete points, $\sum \phi_j(x_i)\phi_k(x_i)$, can be viewed as an inner product $\langle \phi_j, \phi_k \rangle$ of two vectors in a suitably defined vector space. And when the functions (vectors) ϕ_j are **orthogonal**, the inner product $\langle \phi_j, \phi_k \rangle$ is 0 if $j \neq k$, which makes the normal equations trivial to solve. We will discuss details in the next section.

5.3 Orthogonal polynomials and least squares

Our discussion in this section will mostly center around the continuous least squares problem; however, the discrete problem can be approached similarly. Consider the set $C^0[a,b]$, the set of all continuous functions defined on $[a,b]$, and $\mathbf{P_n}$, the set of all polynomials of degree at most n on $[a,b]$. These two sets are vector spaces, the latter a subspace of the former, under the usual operations of function addition and multiplying by a scalar. An inner product on this space is defined as follows: given $f, g \in C^0[a,b]$

$$\langle f, g\rangle = \int_a^b w(x)f(x)g(x)dx \tag{5.12}$$

and the norm of a vector under this inner product is

$$\|f\| = \langle f, f\rangle^{1/2} = \left(\int_a^b w(x)f^2(x)dx\right)^{1/2}.$$

Let's recall the definition of an inner product: it is a real valued function with the following properties:

1. $\langle f, g\rangle = \langle g, f\rangle$
2. $\langle f, f\rangle \geq 0$, with the equality only when $f \equiv 0$
3. $\langle \beta f, g\rangle = \beta \langle f, g\rangle$ for all real numbers β
4. $\langle f_1 + f_2, g\rangle = \langle f_1, g\rangle + \langle f_2, g\rangle$

The mysterious function $w(x)$ in (5.12) is called a **weight function**. Its job is to assign different importance to different regions of the interval $[a,b]$. The weight function is not arbitrary; it has to satisfy some properties.

Definition 87. A nonnegative function $w(x)$ on $[a,b]$ is called a weight function if

1. $\int_a^b |x|^n w(x)dx$ is integrable and finite for all $n \geq 0$
2. If $\int_a^b w(x)g(x)dx = 0$ for some $g(x) \geq 0$, then $g(x)$ is identically zero on (a,b).

With our new terminology and set-up, we can write the least squares problem as follows:

Problem (Continuous least squares) Given $f \in C^0[a,b]$, find a polynomial $P_n(x) \in \mathbf{P_n}$ that minimizes

$$\int_a^b w(x)(f(x) - P_n(x))^2 dx = \langle f(x) - P_n(x), f(x) - P_n(x)\rangle.$$

We will see this inner product can be calculated easily if $P_n(x)$ is written as a linear combination of orthogonal basis polynomials: $P_n(x) = \sum_{j=0}^n a_j\phi_j(x)$.

We need some definitions and theorems to continue with our quest. Let's start with a formal definition of orthogonal functions.

Definition 88. Functions $\{\phi_0, \phi_1, ..., \phi_n\}$ are orthogonal for the interval $[a, b]$ and with respect to the weight function $w(x)$ if

$$\langle \phi_j, \phi_k \rangle = \int_a^b w(x)\phi_j(x)\phi_k(x)dx = \begin{cases} 0 & \text{if } j \neq k \\ \alpha_j > 0 & \text{if } j = k \end{cases}$$

where α_j is some constant. If, in addition, $\alpha_j = 1$ for all j, then the functions are called orthonormal.

How can we find an orthogonal or orthonormal basis for our vector space? Gram-Schmidt process from linear algebra provides the answer.

Theorem 89 (Gram-Schmidt process)**.** *Given a weight function $w(x)$, the Gram-Schmidt process constructs a unique set of polynomials $\phi_0(x), \phi_1(x), ..., \phi_n(x)$ where the degree of $\phi_i(x)$ is i, such that*

$$\langle \phi_j, \phi_k \rangle = \begin{cases} 0 & \text{if } j \neq k \\ 1 & \text{if } j = k \end{cases}$$

and the coefficient of x^n in $\phi_n(x)$ is positive.

Let's discuss two orthogonal polynomials that can be obtained from the Gram-Schmidt process using different weight functions.

Example 90 (Legendre Polynomials)**.** If $w(x) \equiv 1$ and $[a, b] = [-1, 1]$, the first four polynomials obtained from the Gram-Schmidt process, when the process is applied to the monomials $1, x, x^2, x^3, ...$, are:

$$\phi_0(x) = \sqrt{\frac{1}{2}}, \phi_1(x) = \sqrt{\frac{3}{2}}x, \phi_2(x) = \frac{1}{2}\sqrt{\frac{5}{2}}(3x^2 - 1), \phi_3(x) = \frac{1}{2}\sqrt{\frac{7}{2}}(5x^3 - 3x).$$

Often these polynomials are written in its orthogonal form; that is, we drop the requirement $\langle \phi_j, \phi_j \rangle = 1$ in the Gram-Schmidt process, and we scale the polynomials so that the value of each polynomial at 1 equals 1. The first four polynomials in that form are

$$L_0(x) = 1, L_1(x) = x, L_2(x) = \frac{3}{2}x^2 - \frac{1}{2}, L_3(x) = \frac{5}{2}x^3 - \frac{3}{2}x.$$

These are the Legendre polynomials, polynomials we first discussed in Gaussian quadrature, Section 4.3[3]. They can be obtained from the following recursion

$$L_{n+1}(x) = \frac{2n+1}{n+1}xL_n(x) - \frac{n}{n+1}L_{n-1}(x),$$

[3]The Legendre polynomials in Section 4.3 differ from these by a constant factor. For example, in Section 4.3 the third polynomial was $L_2(x) = x^2 - \frac{1}{3}$, but here it is $L_2(x) = \frac{3}{2}(x^2 - \frac{1}{3})$. Observe that multiplying these polynomials by a constant does not change their roots (what we were interested in Gaussian quadrature), or their orthogonality.

$n = 1, 2, \ldots$, and they satisfy

$$\langle L_n, L_n \rangle = \frac{2}{2n+1}.$$

Exercise 5.3-1: Show, by direct integration, that the Legendre polynomials $L_1(x)$ and $L_2(x)$ are orthogonal.

Example 91 (Chebyshev polynomials)**.** If we take $w(x) = (1-x^2)^{-1/2}$ and $[a, b] = [-1, 1]$, and again drop the orthonormal requirement in Gram-Schmidt, we obtain the following orthogonal polynomials:

$$T_0(x) = 1, T_1(x) = x, T_2(x) = 2x^2 - 1, T_3(x) = 4x^3 - 3x, ...$$

These polynomials are called Chebyshev polynomials and satisfy a curious identity:

$$T_n(x) = \cos(n \cos^{-1} x), n \geq 0.$$

Chebyshev polynomials also satisfy the following recursion:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

for $n = 1, 2, \ldots$, and

$$\langle T_j, T_k \rangle = \begin{cases} 0 & \text{if } j \neq k \\ \pi & \text{if } j = k = 0 \\ \pi/2 & \text{if } j = k > 0. \end{cases}$$

If we take the first $n+1$ Legendre or Chebyshev polynomials, call them $\phi_0, ..., \phi_n$, then these polynomials form a basis for the vector space $\mathbf{P_n}$. In other words, they form a linearly independent set of functions, and any polynomial from $\mathbf{P_n}$ can be written as a unique linear combination of them. These statements follow from the following theorem, which we will leave unproved.

Theorem 92. *1. If $\phi_j(x)$ is a polynomial of degree j for $j = 0, 1, ..., n$, then $\phi_0, ..., \phi_n$ are linearly independent.*

2. If $\phi_0, ..., \phi_n$ are linearly independent in $\mathbf{P_n}$, then for any $q(x) \in \mathbf{P_n}$, there exist unique constants $c_0, ..., c_n$ such that $q(x) = \sum_{j=0}^{n} c_j \phi_j(x)$.

Exercise 5.3-2: Prove that if $\{\phi_0, \phi_1, ..., \phi_n\}$ is a set of orthogonal functions, then they must be linearly independent.

We have developed what we need to solve the least squares problem using orthogonal polynomials. Let's go back to the problem statement:

Given $f \in C^0[a,b]$, find a polynomial $P_n(x) \in \mathbf{P_n}$ that minimizes

$$E = \int_a^b w(x)(f(x) - P_n(x))^2 dx = \langle f(x) - P_n(x), f(x) - P_n(x) \rangle$$

with $P_n(x)$ written as a linear combination of orthogonal basis polynomials: $P_n(x) = \sum_{j=0}^n a_j \phi_j(x)$. In the previous section, we solved this problem using calculus by taking the partial derivatives of E with respect to a_j and setting them equal to zero. Now we will use linear algebra:

$$\begin{aligned} E = \left\langle f - \sum_{j=0}^n a_j\phi_j, f - \sum_{j=0}^n a_j\phi_j \right\rangle &= \langle f, f\rangle - 2\sum_{j=0}^n a_j \langle f, \phi_j\rangle + \sum_i \sum_j a_i a_j \langle \phi_i, \phi_j\rangle \\ &= \|f\|^2 - 2\sum_{j=0}^n a_j \langle f, \phi_j\rangle + \sum_{j=0}^n a_j^2 \langle \phi_j, \phi_j\rangle \\ &= \|f\|^2 - 2\sum_{j=0}^n a_j \langle f, \phi_j\rangle + \sum_{j=0}^n a_j^2 \alpha_j \\ &= \|f\|^2 - \sum_{j=0}^n \frac{\langle f, \phi_j\rangle^2}{\alpha_j} + \sum_{j=0}^n \left[\frac{\langle f, \phi_j\rangle}{\sqrt{\alpha_j}} - a_j\sqrt{\alpha_j}\right]^2. \end{aligned}$$

Minimizing this expression with respect to a_j is now obvious: simply choose $a_j = \frac{\langle f,\phi_j\rangle}{\alpha_j}$ so that the last summation in the above equation, $\sum_{j=0}^n \left[\frac{\langle f,\phi_j\rangle}{\sqrt{\alpha_j}} - a_j\sqrt{\alpha_j}\right]^2$, vanishes. Then we have solved the least squares problem! The polynomial that minimizes the error E is

$$P_n(x) = \sum_{j=0}^n \frac{\langle f, \phi_j\rangle}{\alpha_j}\phi_j(x) \tag{5.13}$$

where $\alpha_j = \langle \phi_j, \phi_j\rangle$. And the corresponding error is

$$E = \|f\|^2 - \sum_{j=0}^n \frac{\langle f, \phi_j\rangle^2}{\alpha_j}.$$

If the polynomials $\phi_0, ..., \phi_n$ are orthonormal, then these equations simplify by setting $\alpha_j = 1$.

We will appreciate the ease at which $P_n(x)$ can be computed using this approach, via formula (5.13), as opposed to solving the normal equations of (5.11) when we discuss some examples. But first let's see the other advantage of this approach: how $P_{n+1}(x)$ can be computed from $P_n(x)$. In

Equation (5.13), replace n by $n+1$ to get

$$P_{n+1}(x) = \sum_{j=0}^{n+1} \frac{\langle f, \phi_j \rangle}{\alpha_j} \phi_j(x) = \underbrace{\sum_{j=0}^{n} \frac{\langle f, \phi_j \rangle}{\alpha_j} \phi_j(x)}_{P_n(x)} + \frac{\langle f, \phi_{n+1} \rangle}{\alpha_{n+1}} \phi_{n+1}(x)$$
$$= P_n(x) + \frac{\langle f, \phi_{n+1} \rangle}{\alpha_{n+1}} \phi_{n+1}(x),$$

which is a simple recursion that links P_{n+1} to P_n.

Example 93. Find the least squares polynomial approximation of degree three to $f(x) = e^x$ on $(-1, 1)$ using Legendre polynomials.

Solution. Put $n = 3$ in Equation (5.13) and let ϕ_j be L_j to get

$$P_3(x) = \frac{\langle f, L_0 \rangle}{\alpha_0} L_0(x) + \frac{\langle f, L_1 \rangle}{\alpha_1} L_1(x) + \frac{\langle f, L_2 \rangle}{\alpha_2} L_2(x) + \frac{\langle f, L_3 \rangle}{\alpha_3} L_3(x)$$
$$= \frac{\langle e^x, 1 \rangle}{2} + \frac{\langle e^x, x \rangle}{2/3} x + \frac{\left\langle e^x, \frac{3}{2}x^2 - \frac{1}{2} \right\rangle}{2/5} \left(\frac{3}{2}x^2 - \frac{1}{2} \right) + \frac{\left\langle e^x, \frac{5}{2}x^3 - \frac{3}{2}x \right\rangle}{2/7} \left(\frac{5}{2}x^3 - \frac{3}{2}x \right),$$

where we used the fact that $\alpha_j = \langle L_j, L_j \rangle = \frac{2}{2n+1}$ (see Example 90). We will compute the inner products, which are definite integrals on $(-1, 1)$, using the five-node Gauss-Legendre quadrature we discussed in the previous chapter. The results rounded to four digits are:

$$\langle e^x, 1 \rangle = \int_{-1}^{1} e^x dx = 2.350$$
$$\langle e^x, x \rangle = \int_{-1}^{1} e^x x dx = 0.7358$$
$$\left\langle e^x, \frac{3}{2}x^2 - \frac{1}{2} \right\rangle = \int_{-1}^{1} e^x \left(\frac{3}{2}x^2 - \frac{1}{2} \right) dx = 0.1431$$
$$\left\langle e^x, \frac{5}{2}x^3 - \frac{3}{2}x \right\rangle = \int_{-1}^{1} e^x \left(\frac{5}{2}x^3 - \frac{3}{2}x \right) dx = 0.02013.$$

Therefore

$$P_3(x) = \frac{2.35}{2} + \frac{3(0.7358)}{2} x + \frac{5(0.1431)}{2} \left(\frac{3}{2}x^2 - \frac{1}{2} \right) + \frac{7(0.02013)}{2} \left(\frac{5}{2}x^3 - \frac{3}{2}x \right)$$
$$= 0.1761x^3 + 0.5366x^2 + 0.9980x + 0.9961.$$

Example 94. Find the least squares polynomial approximation of degree three to $f(x) = e^x$ on $(-1, 1)$ using Chebyshev polynomials.

Solution. As in the previous example solution, we take $n = 3$ in Equation (5.13)

$$P_3(x) = \sum_{j=0}^{3} \frac{\langle f, \phi_j \rangle}{\alpha_j} \phi_j(x),$$

but now ϕ_j and α_j will be replaced by T_j, the Chebyshev polynomials, and its corresponding constant; see Example 91. We have

$$P_3(x) = \frac{\langle e^x, T_0 \rangle}{\pi} T_0(x) + \frac{\langle e^x, T_1 \rangle}{\pi/2} T_1(x) + \frac{\langle e^x, T_2 \rangle}{\pi/2} T_2(x) + \frac{\langle e^x, T_3 \rangle}{\pi/2} T_3(x).$$

Consider one of the inner products,

$$\langle e^x, T_j \rangle = \int_{-1}^{1} \frac{e^x T_j(x)}{\sqrt{1-x^2}} dx$$

which is an improper integral due to discontinuity at the end points. However, we can use the substitution $\theta = \cos^{-1} x$ to rewrite the integral as (see Section 4.5)

$$\langle e^x, T_j \rangle = \int_{-1}^{1} \frac{e^x T_j(x)}{\sqrt{1-x^2}} dx = \int_0^{\pi} e^{\cos\theta} \cos(j\theta) d\theta.$$

The transformed integrand is smooth, and it is not improper, and hence we can use composite Simpson's rule to estimate it. The following estimates are obtained by taking $n = 20$ in the composite Simpson's rule:

$$\begin{aligned}
\langle e^x, T_0 \rangle &= \int_0^{\pi} e^{\cos\theta} d\theta = 3.977 \\
\langle e^x, T_1 \rangle &= \int_0^{\pi} e^{\cos\theta} \cos\theta d\theta = 1.775 \\
\langle e^x, T_2 \rangle &= \int_0^{\pi} e^{\cos\theta} \cos 2\theta d\theta = 0.4265 \\
\langle e^x, T_3 \rangle &= \int_0^{\pi} e^{\cos\theta} \cos 3\theta d\theta = 0.06964
\end{aligned}$$

Therefore

$$\begin{aligned}
P_3(x) &= \frac{3.977}{\pi} + \frac{3.55}{\pi} x + \frac{0.853}{\pi}(2x^2 - 1) + \frac{0.1393}{\pi}(4x^3 - 3x) \\
&= 0.1774x^3 + 0.5430x^2 + 0.9970x + 0.9944.
\end{aligned}$$

Python code for orthogonal polynomials

Computing Legendre polynomials

Legendre polynomials satisfy the following recursion:

$$L_{n+1}(x) = \frac{2n+1}{n+1} x L_n(x) - \frac{n}{n+1} L_{n-1}(x)$$

for $n = 1, 2, \ldots$, with $L_0(x) = 1$, and $L_1(x) = x$.

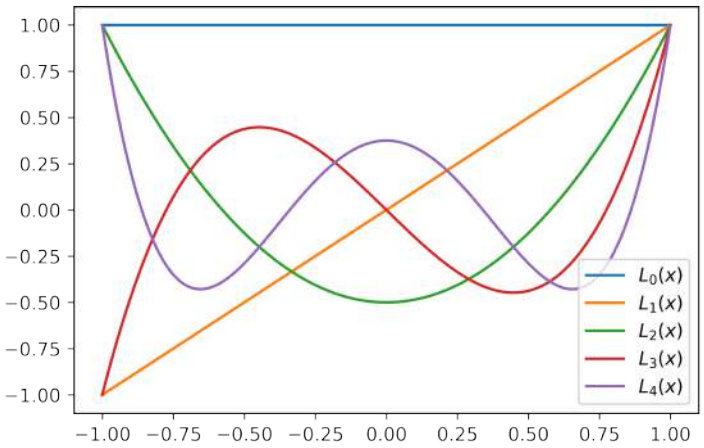
The Python code implements this recursion, with a little modification: the index $n+1$ is shifted down to n, so the modified recursion is: $L_n(x) = \frac{2n-1}{n} x L_{n-1}(x) - \frac{n-1}{n} L_{n-2}(x)$, for $n = 2, 3, \ldots$.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [2]: def leg(x, n):
            if n == 0:
                return 1
            elif n == 1:
                return x
            else:
                return ((2*n-1)/n)*x*leg(x,n-1)-((n-1)/n)*leg(x,n-2)
```

Here is a plot of the first five Legendre polynomials:

```
In [3]: xaxis = np.linspace(-1, 1, 200)
        legzero = np.array(list(map(lambda x: leg(x,0), xaxis)))
        legone = leg(xaxis, 1)
        legtwo = leg(xaxis, 2)
        legthree = leg(xaxis, 3)
        legfour = leg(xaxis, 4)
        plt.plot(xaxis, legzero, label='$L_0(x)$')
        plt.plot(xaxis, legone, label='$L_1(x)$')
        plt.plot(xaxis, legtwo, label='$L_2(x)$')
        plt.plot(xaxis, legthree, label='$L_3(x)$')
        plt.plot(xaxis, legfour, label='$L_4(x)$')
        plt.legend(loc='lower right');
```



Least-squares using Legendre polynomials

In Example 93, we computed the least squares approximation to e^x using Legendre polynomials. The inner products were computed using the five-node Gauss-Legendre rule below.

```
In [4]: def gauss(f):
            return 0.2369268851*f(-0.9061798459) + 0.2369268851*f(0.9061798459) + \
                   0.5688888889*f(0) + 0.4786286705*f(0.5384693101) + \
                   0.4786286705*f(-0.5384693101)
```

The inner product, $\left\langle e^x, \frac{3}{2}x^2 - \frac{1}{2} \right\rangle = \int_{-1}^{1} e^x(\frac{3}{2}x^2 - \frac{1}{2})dx$ is computed as

```
In [5]: gauss(lambda x: ((3/2)*x**2-1/2)*np.exp(x))
```

```
Out[5]: 0.1431256282441218
```

Now that we have a code **leg(x,n)** that generates the Legendre polynomials, we can do the above computation without explicitly specifying the Legendre polynomial. For example, since $\frac{3}{2}x^2 - \frac{1}{2} = L_2$, we can apply the **gauss** function directly to $L_2(x)e^x$:

```
In [6]: gauss(lambda x: leg(x,2)*np.exp(x))
```

```
Out[6]: 0.14312562824412176
```

The following function **polyLegCoeff(f,n)** computes the coefficients $\frac{\langle f,L_j \rangle}{\alpha_j}$ of the least squares polynomial $P_n(x) = \sum_{j=0}^{n} \frac{\langle f,L_j \rangle}{\alpha_j} L_j(x), j = 0, 1, ..., n$, for any f and n, where L_j are the Legendre polynomials. The coefficients are the outputs that are returned when the function is finished.

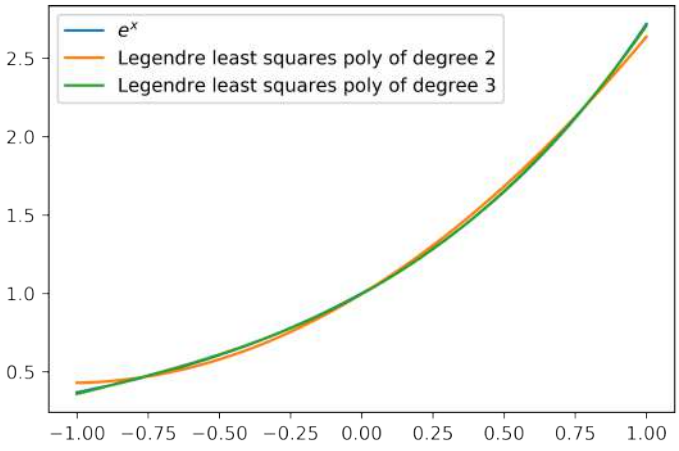
```
In [7]: def polyLegCoeff(f, n):
            A = np.zeros(n+1)
            for j in range(n+1):
                A[j] = gauss(lambda x: leg(x,j)*f(x))*(2*j+1)/2
            return A
```

Once the coefficients are computed, evaluating the polynomial can be done efficiently by using the coefficients. The next function **polyLeg(x,n,A)** evaluates the least squares polynomial $P_n(x) = \sum_{j=0}^{n} \frac{\langle f,L_j\rangle}{\alpha_j} L_j(x), j = 0, 1, ..., n$, where the coefficients $\frac{\langle f,L_j\rangle}{\alpha_j}$, stored in A, are obtained from calling the function **polyLegCoeff(f,n)**.

```
In [8]: def polyLeg(x, n, A):
            sum = 0.
            for j in range(n+1):
                sum += A[j]*leg(x, j)
            return sum
```

Here we plot $y = e^x$ together with its least squares polynomial approximations of degree two and three, using Legendre polynomials. Note that every time the function **polyLegCoeff** is evaluated, a new coefficient array **A** is obtained.

```
In [9]: xaxis = np.linspace(-1, 1, 200)
        A = polyLegCoeff(lambda x: np.exp(x), 2)
        deg2 = polyLeg(xaxis, 2, A)
        A = polyLegCoeff(lambda x: np.exp(x), 3)
        deg3 = polyLeg(xaxis, 3, A)
        plt.plot(xaxis, np.exp(xaxis), label='$e^x$')
        plt.plot(xaxis, deg2, label='Legendre least squares poly of degree 2')
        plt.plot(xaxis, deg3, label='Legendre least squares poly of degree 3')
        plt.legend(loc='upper left');
```



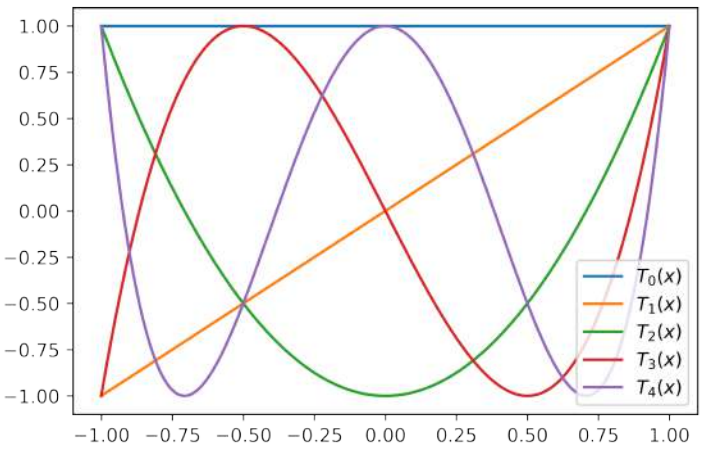
Computing Chebyshev polynomials

The following function implements the recursion Chebyshev polynomials satisfy: $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$, for $n = 1, 2, ...$, with $T_0(x) = 1$ and $T_1(x) = x$. Note that in the code the index is shifted from $n + 1$ to n.

```
In [10]: def cheb(x, n):
             if n == 0:
                 return 1
             elif n == 1:
                 return x
             else:
                 return 2*x*cheb(x,n-1)-cheb(x,n-2)
```

Here is a plot of the first five Chebyshev polynomials:

```
In [11]: xaxis = np.linspace(-1, 1, 200)
         chebzero = np.array(list(map(lambda x: cheb(x,0), xaxis)))
         chebone = cheb(xaxis, 1)
         chebtwo = cheb(xaxis, 2)
         chebthree = cheb(xaxis, 3)
         chebfour = cheb(xaxis, 4)
         plt.plot(xaxis, chebzero, label='$T_0(x)$')
         plt.plot(xaxis, chebone, label='$T_1(x)$')
         plt.plot(xaxis, chebtwo, label='$T_2(x)$')
         plt.plot(xaxis, chebthree, label='$T_3(x)$')
         plt.plot(xaxis, chebfour, label='$T_4(x)$')
         plt.legend(loc='lower right');
```



Least squares using Chebyshev polynomials

In Example 94, we computed the least squares approximation to e^x using Chebyshev polynomials. The inner products, after a transformation, were computed using the composite Simpson rule. Below is the Python code for the composite Simpson rule we discussed in the previous chapter.

```
In [12]: def compsimpson(f, a, b, n):
             h = (b-a)/n
             nodes = np.zeros(n+1)
             for i in range(n+1):
                 nodes[i] = a+i*h
             sum = f(a) + f(b)
             for i in range(2, n-1, 2):
                 sum += 2*f(nodes[i])
             for i in range(1, n, 2):
                 sum += 4*f(nodes[i])
             return sum*h/3
```

The integrals in Example 94 were computed using the composite Simpson rule with $n = 20$. For example, the second integral $\langle e^x, T_1 \rangle = \int_0^\pi e^{\cos\theta} \cos\theta d\theta$ is computed as:

```
In [13]: compsimpson(lambda x: np.exp(np.cos(x))*np.cos(x), 0, np.pi, 20)
```

```
Out[13]: 1.7754996892121808
```

Next we write two functions, **polyChebCoeff(f,n)** and **polyCheb(x,n,A)**. The first function computes the coefficients $\frac{\langle f,T_j \rangle}{\alpha_j}$ of the least squares polynomial $P_n(x) = \sum_{j=0}^{n} \frac{\langle f,T_j \rangle}{\alpha_j} T_j(x), j = 0, 1, ..., n$, for any f and n, where T_j are the Chebyshev polynomials. The coefficients are returned as the output of the first function.

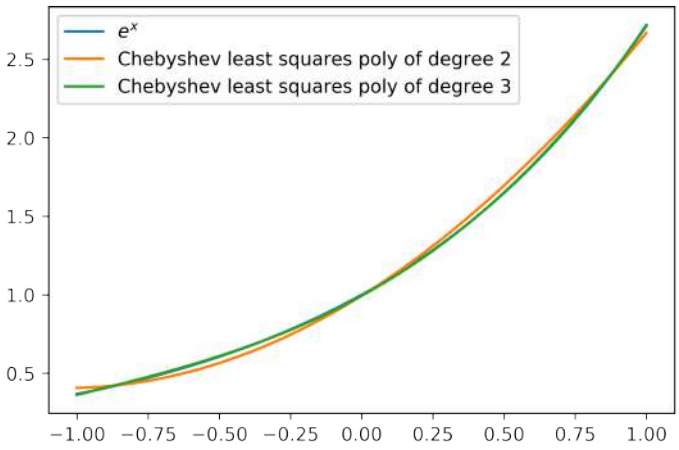
The integral $\langle f, T_j \rangle$ is transformed to the integral $\int_0^\pi f(\cos\theta)\cos(j\theta)d\theta$, similar to the derivation in Example 94, and then the transformed integral is computed using the composite Simpson's rule by **polyChebCoeff**.

```
In [14]: def polyChebCoeff(f, n):
             A = np.zeros(n+1)
             A[0] = compsimpson(lambda x: f(np.cos(x)), 0, np.pi, 20)/np.pi
             for j in range(1, n+1):
                 A[j] = compsimpson(lambda x: f(np.cos(x))*np.cos(j*x), 0, np.pi, 20)*2/np.pi
             return A
```

```
In [15]: def polyCheb(x, n, A):
             sum = 0.
             for j in range(n+1):
                 sum += A[j]*cheb(x, j)
             return sum
```

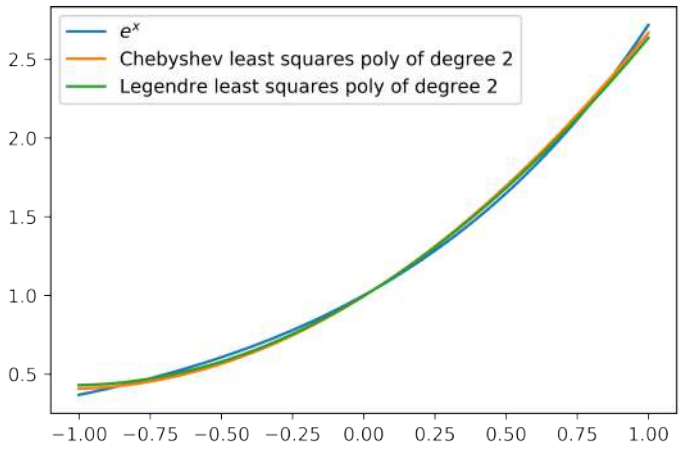
Next we plot $y = e^x$ together with polynomial approximations of degree two and three using Chebyshev basis polynomials.

```
In [16]: xaxis = np.linspace(-1, 1, 200)
         A = polyChebCoeff(lambda x: np.exp(x), 2)
         deg2 = polyCheb(xaxis, 2, A)
         A = polyChebCoeff(lambda x: np.exp(x), 3)
         deg3 = polyCheb(xaxis, 3, A)
         plt.plot(xaxis, np.exp(xaxis), label='$e^x$')
         plt.plot(xaxis, deg2, label='Chebyshev least squares poly of degree 2')
         plt.plot(xaxis, deg3, label='Chebyshev least squares poly of degree 3')
         plt.legend(loc='upper left');
```



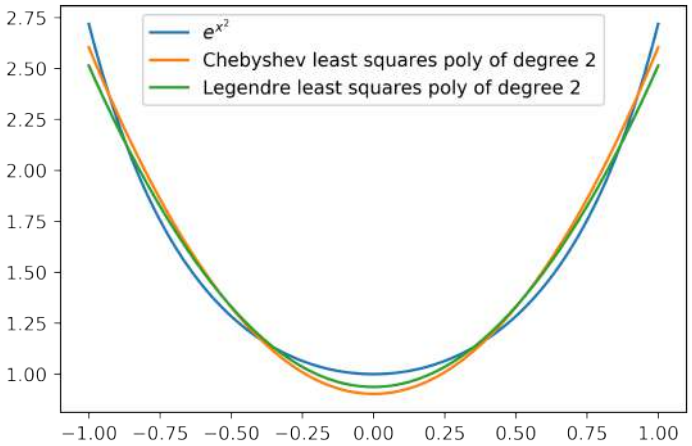
The cubic Legendre and Chebyshev approximations are difficult to distinguish from the function itself. Let's compare the quadratic approximations obtained by Legendre and Chebyshev polynomials. Below, you can see visually that Chebyshev does a better approximation at the end points of the interval. Is this expected?

```
In [17]: xaxis = np.linspace(-1, 1, 200)
         A = polyChebCoeff(lambda x: np.exp(x), 2)
         cheb2 = polyCheb(xaxis, 2, A)
         A = polyLegCoeff(lambda x: np.exp(x), 2)
         leg2 = polyLeg(xaxis, 2, A)
         plt.plot(xaxis, np.exp(xaxis), label='$e^x$')
         plt.plot(xaxis, cheb2, label='Chebyshev least squares poly of degree 2')
         plt.plot(xaxis, leg2, label='Legendre least squares poly of degree 2')
         plt.legend(loc='upper left');
```



In the following, we compare second degree least squares polynomial approximations for $f(x) = e^{x^2}$. Compare how good the Legendre and Chebyshev polynomial approximations are in the midinterval and toward the endpoints.

```
In [18]: f = lambda x: np.exp(x**2)
         xaxis = np.linspace(-1, 1, 200)
         A = polyChebCoeff(lambda x: f(x), 2)
         cheb2 = polyCheb(xaxis, 2, A)
         A = polyLegCoeff(lambda x: f(x), 2)
         leg2 = polyLeg(xaxis, 2, A)
         plt.plot(xaxis, f(xaxis), label='$e^{x^2}}$')
         plt.plot(xaxis, cheb2, label='Chebyshev least squares poly of degree 2')
         plt.plot(xaxis, leg2, label='Legendre least squares poly of degree 2')
         plt.legend(loc='upper center');
```



Exercise 5.3-3: Use Python to compute the least squares polynomial approximations $P_2(x), P_4(x), P_6(x)$ to $\sin 4x$ using Chebyshev basis polynomials. Plot the polynomials together with $\sin 4x$.

References

[1] Abramowitz, M., and Stegun, I.A., 1965. Handbook of mathematical functions: with formulas, graphs, and mathematical tables (Vol. 55). Courier Corporation.

[2] Chace, A.B., and Manning, H.P., 1927. The Rhind Mathematical Papyrus: British Museum 10057 and 10058. Vol 1. Mathematical Association of America.

[3] Atkinson, K.E., 1989. An Introduction to Numerical Analysis, Second Edition, John Wiley & Sons.

[4] Burden, R.L, Faires, D., and Burden, A.M., 2016. Numerical Analysis, 10th Edition, Cengage.

[5] Capstick, S., and Keister, B.D., 1996. Multidimensional quadrature algorithms at higher degree and/or dimension. Journal of Computational Physics, 123(2), pp.267-273.

[6] Chan, T.F., Golub, G.H., and LeVeque, R.J., 1983. Algorithms for computing the sample variance: Analysis and recommendations. The American Statistician, 37(3), pp.242-247.

[7] Cheney, E.W., and Kincaid, D.R., 2012. Numerical mathematics and computing. Cengage Learning.

[8] Glasserman, P., 2013. Monte Carlo methods in Financial Engineering. Springer.

[9] Goldberg, D., 1991. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys (CSUR), 23(1), pp.5-48.

[10] Heath, M.T., 1997. Scientific Computing: An introductory survey. McGraw-Hill.

[11] Higham, N.J., 1993. The accuracy of floating point summation. SIAM Journal on Scientific Computing, 14(4), pp.783-799.

[12] Isaacson, E., and Keller, H.B., 1966. Analysis of Numerical Methods. John Wiley & Sons.

Index