MATH2105 Scientific Computation. Lecture Notes

Dr. Marco Iglesias

School of Mathematical Sciences

University of Nottingham

Table of contents

Welcome!

These are the Lecture Notes of MATH2105 Scientific Computation. In this section welcome page we discuss the general aspects of the module.

Weekly schedule

Table 1: Weekly schedule 2024-2025.

Week	Date	Section	Topics
1	27/1/25	Chapter 1	• Lecture 1: – Section 1.2
1	29/1/25	Chapter 1 Chapter 1	• Lecture 1: – Section 1.2 • Lecture 2: – Section 1.3
2	3/2/25	Chapter 1	• Lecture 2: – Section 1.3
2	5/2/25	Chapter 1	• Lecture 3: – Section 1.4 – Section 1.5 – Section 1.6
3	10/2/25	Chapter 2	• Lecture 4: – Section 2.1
3	12/2/25	Chapter 2	• Lecture 5: – Section 2.2
4	17/2/25	Chapter 2	• Lecture 6: – Section 2.3
4	19/2/25	Chapter 3 **CW 1 OUT**	• Lecture 7: – Section 3.1
5	24/2/25	Chapter 3	• Lecture 8: – Section 3.2
5	26/2/25	Chapter 3	• Lecture 9: – Section 3.3
6	26/2/25	Chapter 4	• Lecture 10:
6	3/3/25	Chapter 4 **CW 1 DUE**	• Lecture 11:
7	5/3/25	Chapter 4	• Lecture 12:
7	10/3/25	Chapter 5	• Lecture 13:
8	12/3/25	Chapter 5	• Lecture 14:
8	17/3/25	Chapter 5	• Lecture 15:
9	19/3/25	Chapter 6	• Lecture 16:
9	24/3/25	Chapter 6	• Lecture 17:

Introduction to the Module

This course aims to introduce the concept of numerical approximation to problems that cannot be solved analytically, and to develop skills in Python through implementation of numerical methods. This course provides an important foundation on which students can develop skills and understanding in computational applied mathematics.

Summary of Content

This course introduces basic techniques in numerical methods and numerical analysis which can be used to generate approximate solutions to problems that may not be amenable to analytical techniques. Specific topics include:

- Chapter 1 (bisection method, fixed-point iteration, Newton's method, convergence);
- Linear systems of equations: Chapter 2 (Gaussian elimination, operation count, pivoting strategies, matrix factorisation, special matrices: diagonally dominant, symmetric positive definite);
- Linear systems of equations: Chapter 3 (matrix norms, Jacobi & Gauss-Seidel method, convergence);
- Chapter 4 (Lagrange polynomials, Lagrange form, error analy- sis);
- Chapter 5 (difference formulae, numerical quadrature: trapezoidal, Simpson & midpoint rule, composite rules, Richardson extrapolation);
- Chapter 6 (Euler's method, wellposedness of IVPs, higher-order RK methods, local truncation error);
- Implementing algorithms in Python, basic elements of finite digit arithmetic.

Learning Outcomes

A student who completes this course successfully will be able to:

- Solve linear equations(approximately) using root finding methods,and analyse their convergence;
- Solve linear systems of equations using direct methods,and analyse their computational complexity;
- Solve linear systems of equations using iterative techniques, and analyse their convergence;
- Approximate functions by polynomial interpolants, and analyse their accuracy;
- Approximate derivatives and definite integrals using numerical differentiation and integration, and analyse their convergence;
- Approximate ODEs using numerical methods, and analyse their convergence;
- Implement reusable codes in Python.

Study Material

- **Lecture Notes** (this document): These contain all the material that will be covered in the module. **All this material can be assessed**. Most sections of the Lecture Notes have a section of Additional Exercises (and solutions) as well as Computer Activities. You are encouraged to work on the Computer Activities during the drop-in sessions.
- **Lecture Slides**: These are used during lectures and have a comprised version of the material in the lecture notes.
- We follow closely **some** of the chapters from Burden, Faires & Burden 2016 (10th ed.)

Assessment and reassessment

- Courseworks: CW1, CW2, each worth 20% of final mark
- Exam: Written exam 60% of final mark

Reassessment is by a new opportunity to take the final exam only. This holds both for resits taken in August/September and for resits taken in January/May, even when the January/May resits are "resits with attendance." The module re- sit mark is computed by replacing fail range marks (40% for undergraduate students, 50% for postgraduate students) in each of the individual assessed components of the module by the resit final exam mark if higher. Students with accepted extenuating circumstances claims relating to any of the assessed components should refer to their extenuating circumstances decision letter

Python requirements

You are expected to have basic familiarity with Python, in particular: logic and loops, functions, NumPy and matplotlib.pyplot. We recommend that you review Core Python Notes

In this module (including courseworks) it will always be assumed that the package `numpy` is imported as `np`, and `matplotlib.pyplot` as `plt`.

We strongly encourage you to use Spyder IDE (Integrated Development Environment).

How and Where to run Spyder:

Spyder comes as part of the Anaconda package. Here are three options on how to run Spyder:

1. You can choose to install Anaconda on you personal device (if not done already). See also Anaconda's website.
2. You can open Anaconda on any University of Nottingham computer.
3. You can open a UoN Virtual Desktop on your own personal device, which is virtually the same as logging onto a University of Nottingham computer, but through the virtual desktop. The simply open Anaconda.

The A18 Computer Room in the Mathematical Sciences building has a number of computers available, as well as desks with dual monitors that you can plug into your own laptop.

Useful links

- Python 3 Online Documentation
- NumPy User Guide
- Matplotlib Quick Start Guide
- Python like you mean it

Chapter 1

Nonlinear Equations

1.1 Introduction

This section delves into several numerical techniques for approximating the solutions of nonlinear equations involving a single variable. The need to study these methods stems from the reality that most nonlinear equations cannot be solved analytically. For instance, consider the following nonlinear equation:

$$\cos(p) - p^3 = 0 \tag{1.1}$$

This equation does not have an analytic solution, although we can prove that a solution (or root) does indeed exist. In other words, there is some $p \in \mathbb{R}$ that satisfies the equation.

Note that we can write equation (Equation 1.1) in the following form $f(p) = 0$ where $f(p) = \cos(p) - p^3$. Also, while the function f can be defined for all $p \in \mathbb{R}$, we may be interested in approximating a solution to the nonlinear equation in given interval $[a, b]$. This motivates the following definition.

Definition 1.1 (Root-finding problem)**.** Given a function $f : [a, b] \to \mathbb{R}$ the root-finding problem consists of finding $p \in (a, b)$ such that

$$f(p) = 0.$$

The solution p is called a root or zero of f.

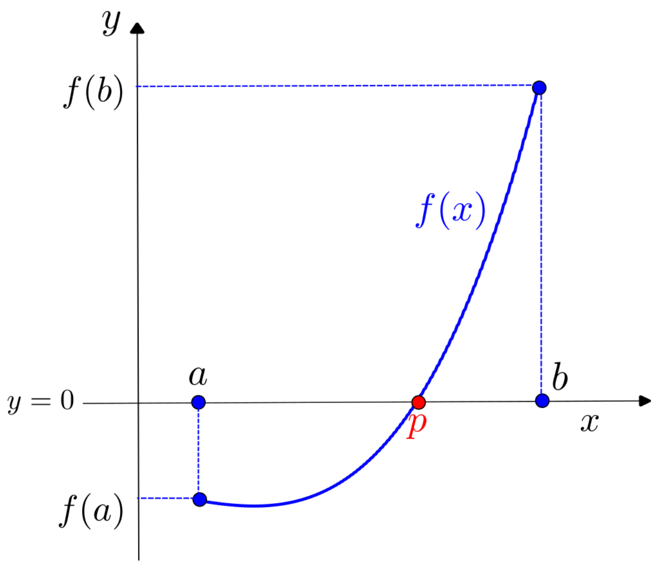


Figure 1.1: Root-finding problem

In the following sections we will introduce algorithms aimed at solving the root-finding problem. It is important, however, to ensure that the problem has a solution.

Theorem 1.1 (Existence of a root). *Given $f : [a,b] \to \mathbb{R}$ continuous on $[a,b]$ with $f(a)f(b) < 0$ there is $p \in (a,b)$ such that $f(p) = 0$*

Proof. Apply the Intermediate Value Theorem (Theorem A.1) for the case $K = 0$. In this case $f(a)$ has oposite sign to $f(b)$. Thus the problem statement is guaranteed to have a solution. ☐

Note

Note that for Theorem 1.1 we need the following **assumptions:**

- $f : [a,b] \to \mathbb{R}$ continuous. We denote this by $f \in C[a,b]$ (see Definition A.3), and
- $f(a)f(b) < 0$: this means that $f(a)$ and $f(b)$ have opposite sign.

1.2 Bisection Method

1.2.1 Main Idea

The Bisection method is perhaps the most straightforward algorithm that we can employ to solve a root-finding problem. The idea of the method is to iteratively bisect the interval $[a,b]$ into subintervals $[a,b] = [a_1,b_1] \supset [a_2,b_2] \supset ... \supset [a_n,b_n] \supset ...$ so that the midpoint of each subinterval converges to a root of f. More specifically, consider the following steps:

Step 0: Set $a_1 = a$, $b_1 = b$.

Step 1: Bisect the interval $[a_1,b_1]$, i.e. compute the midpoint $p_1 = \frac{1}{2}(a_1 + b_1)$ and evaluate $f(p_1)$ (Figure 1.2).

Step 2: (Update Interval): If $f(p_1)f(a_1) > 0$ i.e. $f(p_1)$ has the same sign of $f(a_1)$, then set $a_2 = p_1$ and $b_2 = b_1$, otherwise then set $a_2 = a_1$ and $b_2 = p_1$ (Figure 1.3). Step 1 and Step 2 is what we call an iteration.

Repeat Steps 1 and 2 with the new interval $[a_2,b_2]$, i.e. compute midpoint p_2 and update interval $[a_3,b_3]$ (Figure 1.4 and Figure 1.5).

Carry on!: Continue until p_n is a zero of f i.e. $f(p_n) = 0$. (Figure 1.6)

Let us try this:

Exercise 1.1. Use the bisection method to find p_1, p_2, p_3 for

$$f(x) = x^3 + x^2 - 2x - 2$$

and $[a,b] = [1,2]$. Note that the exact solution is $x = \sqrt{2} \approx 1.41421$.

Solution

```
#define function via lambda function:
f = lambda x: x**3+x**2-2*x-2

#initial interval end points:
a, b = 1.0, 2.0

#initial function values:
print( "f(a)=",f(a), "f(b)=", f(b),  )
```

```
f(a)= -2.0 f(b)= 6.0
```

```
#Check they have different signs:
print( "f(a)f(b):",f(a)*f(b)  )
```

```
f(a)f(b): -12.0
```

```
#alternatively:
print( "is f(a)f(b)<0?", f(a)*f(b)<0 )
```

```
is f(a)f(b)<0? True
```

Hence $f(a)$ and $f(b)$ have different sign and since f is continuous it has a root so let's apply the bisection method. Note that in the code above we defined f as a Lambda function.

First iteration:

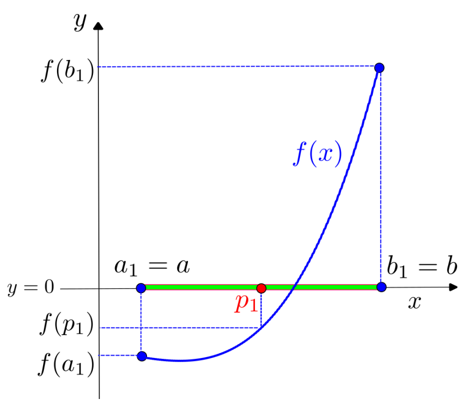


Figure 1.2: Step 1

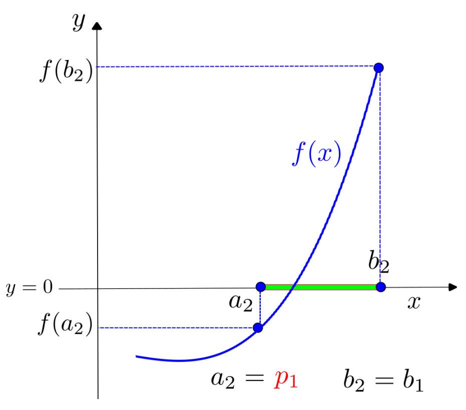


Figure 1.3: Step 2

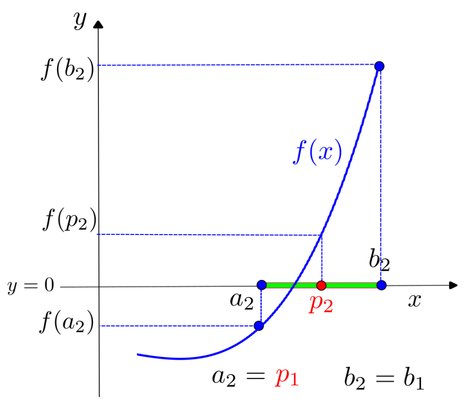


Figure 1.4: Repeat Step 1

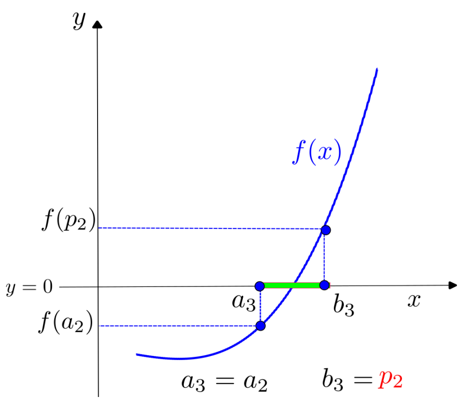


Figure 1.5: Repeat Step 2

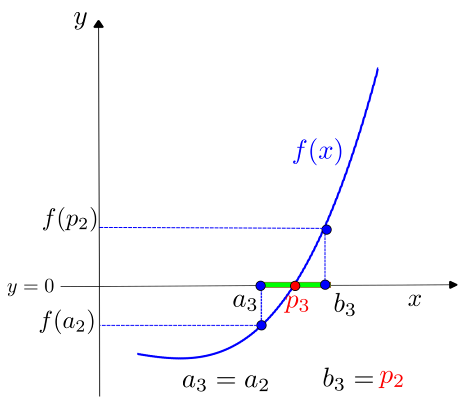


Figure 1.6: Carry on

```
#define interval [a1,b1]=[a,b]
a1, b1 = a , b

p1= (a1+b1)/2 #midpoint

fp1=f(p1) #evaluate

print("p1=", p1)
```

```
p1= 1.5
```

```
print("f(p1)=", fp1)
```

```
f(p1)= 0.625
```

```
#Check sign
print( "is f(a1)f(p1)>0?",  f(a1)*fp1>0 )
```

```
is f(a1)f(p1)>0? False
```

```

#Update interval:
if f(a1)*fp1>0: #same sign
    a2 , b2 = p1 , b1
else:
    a2 , b2= a1, p1

```

Second iteration:

```
#New Interval end points:
print("a2=", a2,"b2=", b2)
```

```
a2= 1.0 b2= 1.5
```

```
p2=(a2+b2)/2 #midpoint
fp2=f(p2) #evaluate

print("p2=", p2)
```

```
p2= 1.25
```

```
print("f(p2)=", fp2)
```

```
f(p2)= -0.984375
```

```

#Update Interval
if f(a2)*fp2>0: #same sign
    a3 , b3 = p2 , b2
else:
    a3 , b3= a2, p2
```

Third Iteration:

```
#New Interval end points:
print("a3=", a3,"b3=", b3)
```

```
a3= 1.25 b3= 1.5
```

```
p3=(a3+b3)/2 #midpoint

fp3=f(p3) #evaluate

print("p3=", p3)
```

```
p3= 1.375
```

```
print("f(p3)=", fp3)
```

```
f(p3)= -0.259765625
```

Hence the final answer is

```
print("p1=" ,p1,"p2=",p2,"p3=", p3)
```

```
p1= 1.5 p2= 1.25 p3= 1.375
```

Midpoints and intervals are shown in Figure 1.7

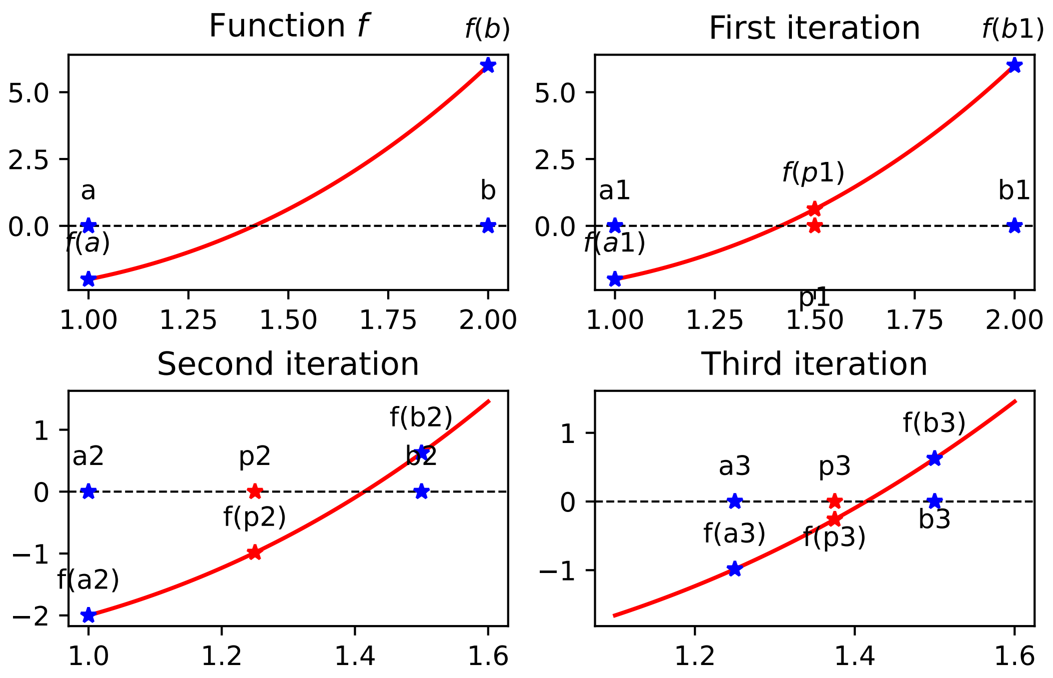


Figure 1.7: Solution Exercise 1.1

1.2.2 Pseudocode and Python code

To provide a helpful link between the mathematical explanation of an algorithm—expressed through words and formulas—and its implementation as executable code, these notes often represent algorithms in pseudocode. This pseudocode combines descriptive language, mathematical expressions, and a style of notation that bears some similarity to code in languages like Python.

Using pseudocode is advantageous over jumping directly into writing code in a specific language (such as Python), as it makes transitioning to other programming languages easier if you choose to implement the algorithm differently in the future.

We can write a simple version of the pseudocode for the bisection method in Algorithm 1.1 .

Algorithm 1.1 Bisection (simple)

```
1: Inputs:
2:     $f \in C[a,b]$ with $f(a)f(b) < 0$,
3:     $N_{\max}$: max. no. of iterations.
4: Output: $p$: approx. to the root of $f$.
5: procedure BisectionSimple($f, a, b, N_{max}$)
6:     Set $f_a = f(a)$
7:     for $n = 1$ to $N_{max}$ do
8:         Compute midpoint $p = (a+b)/2$ and $f_p = f(p)$
9:         if $f_a \cdot f_p > 0$ then
10:            set $a = p$ and $f_a = f_p$
11:        else
12:            set $b = p$ ($f_a$ is unchanged)
13:        end if
14:    end for
15: end procedure
```

A simple python code to implement Algorithm 1.1 is given in Listing 1.1. We have used the function $f(x) = x^3 + x^2 - 2x - 2$ on the interval $[1, 2]$ which has a root $p = \sqrt{2}$ (see Exercise 1.1).

Listing 1.1 `warming_up_bisect.py` Python implementation of Algorithm 1.1

```python
f = lambda x: x**3+x**2-2*x-2

a = 1.0
b = 2.0
Nmax = 4 #maximum number of iterations

#Begin bisection method:
fa=f(a)
for n in range(Nmax):
    p=(a+b)/2  # midpoint
    fp=f(p)    # evaluate f at midpoint
    #define new interval
    if fp*fa>0:
        a=p
        fa=fp
    else:
        b=p
    print(f"iteration n= {n+1} ====> p_n:{p:.15f}")
```

```
iteration n= 1 ====> p_n:1.500000000000000
iteration n= 2 ====> p_n:1.250000000000000
iteration n= 3 ====> p_n:1.375000000000000
iteration n= 4 ====> p_n:1.437500000000000
```

! Important 1: Computing Activity

Create a file `warming_up_bisect.py` with the code from Listing 1.1. Have a go at running this file with different choices of `Nmax` and verify whether the algorithm converges to the root of f.

The simple code from Listing 1.1 is not very useful if we wanted to reuse the code and test it different functions f or call it within another script. Instead, we would prefer to have the bisection method implemented as a function that we can call as many times as we want for different inputs (f, a, b, N_{max}) or to import it as module on a different script. We address this limitation in the code of Listing 1.2 where we define a function `bisection_simple(f,a,b,Nmax)` using the keyword `def` that performs the bisection method using the inputs (f, a, b, N_{max}). The function is to return a `numpy.ndarray` called `p_array`, shape `(Nmax, )`, which is a 1-D array of the approximations p_n $(n = 1, 2, \ldots, N_{max})$ computed by the bisection method. Of course, to work with `numpy` arrays we first need to import (see top of Listing 1.2) the python library `numpy`

Note that before we proceed to test the function `bisection_simple(f,a,b,Nmax)` with a specific set of inputs, we have included the statement `if __name__ == "__main__":`. If we save Listing 1.2 into a file named `bisec_func.py` and we run it, the code below the statement `if __name__ == "__main__":` will be executed. However, if we import `bisec_func.py` as a module within another script the code after the `if __name__ == "__main__":` statement wil not executed. You will practice how to import modules in Important 2.

i Note 1: Docstrings

The text included within the triple quotations `"""` is called Docstring which allows us to provide documentation of a module (and its functions). We can retrieve the dosctring as follows:

```python
help(bisection_simple)
```

```
Help on function bisection_simple in module __main__:

bisection_simple(f, a, b, Nmax)
    Bisection Method: Returns a numpy array of the
    sequence of approximations obtained by the bisection method.

    Inputs:
    ----------
    f : function
        Input function for which the zero is to be found.
    a : float
        Left side of interval.
    b : float
        Right side of interval.
    Nmax : integer
        Number of iterations to be performed.

    Returns
    -------
    p_array : numpy.ndarray,
            Array containing the sequence of approximations.
            The shape is (Nmax,)
```

1.2.3 Convergence

Can we guarantee that the algorithm converges? In other words, can we guarantee that

$$\lim_{n\to\infty} p_n = p$$

for some p that satisfies $f(p) = 0$ (i.e. p is a root of f).

Indeed, under the appropriate conditions, convergence is ensured by the following:

Theorem 1.2 (Error bound for the Bisection method)**.** *Suppose $f \in C[a,b]$ and $f(a)f(b) < 0$, the bisection method generates a sequence $\{p_n\}_{n=1}^{\infty}$ approximating a root of f, denoted by p, with*

$$|p_n - p| \leq \frac{b-a}{2^n}, \qquad \textit{when}, \quad n \geq 1$$

Proof. At the nth iteration ($n \geq 1$) we have the interval $[a_n, b_n]$ and $p_n = \frac{1}{2}(a_n + b_n)$ its midpoint. Without loss of generality suppose that the root p is in $[a_n, p_n]$. Therefore (see Figure 1.8):

$$|p_n - p| \leq |p_n - a_n|$$

Note that $|p_n - a_n| = \frac{1}{2}(b_n - a_n)$ since $\frac{1}{2}(b_n - a_n)$ is the length of the interval $[a_n, p_n]$. Therefore:

$$|p_n - p| \leq \frac{1}{2}(b_n - a_n) = \frac{1}{2}\left[\frac{1}{2}(b_{n-1} - a_{n-1})\right]$$

where we have used that the length of $[a_{n-1}, b_{n-1}]$ is half the length of $[a_n, b_n]$. In fact, recursively we have:

$$\frac{1}{2}\left[\frac{1}{2}(b_{n-1} - a_{n-1})\right] = \frac{1}{2}\frac{1}{2^2}\left[(b_{n-2} - a_{n-2})\right] = \cdots = \frac{1}{2}\frac{1}{2^{n-1}}\left[(b_1 - a_1)\right] = \frac{1}{2^n}(b-a)$$

and thus,

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

which completes the proof. □

Note

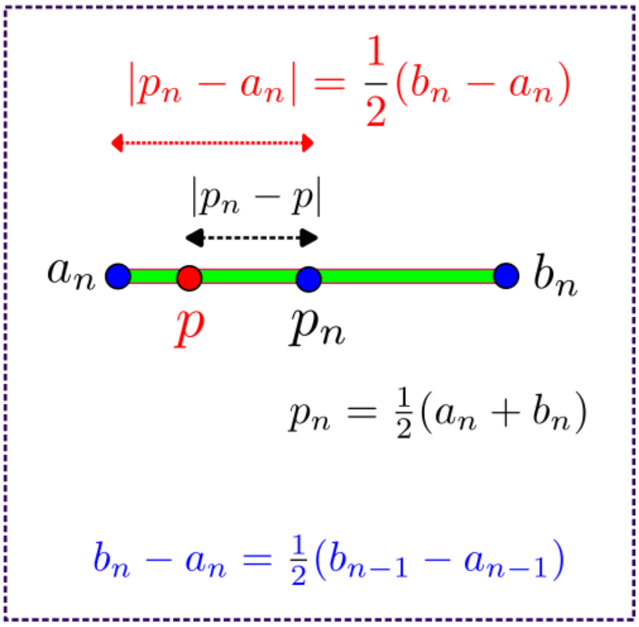
Note that Theorem 1.2 also tells us how accurate the method is at every iteration:

$$\text{Absolute Error}: \; E_n := \; |p_n - p| \leq \frac{1}{2}(b_n - a_n) = \frac{b-a}{2^n} \tag{1.2}$$

Hence, we can compute the number of iterations required to achieve a desired level of accuracy (in terms of absolute error). We can use this information to stop the bisection algorithm.

Exercise 1.2.

1. Find an expression for the minimum number of iterations N necessary for the Bisection Method to solve $f(x) = 0$ with a accuracy `TOL`. Assume $f \in C[a,b]$ and $f(a)f(b) < 0$.
2. Choose $a = 1$, $b = 2$ and `TOL` $= 10^{-3}$ and find N.



$|p_n - a_n| = \frac{1}{2}(b_n - a_n)$
$|p_n - p|$
a_n
b_n
p
p_n
$p_n = \frac{1}{2}(a_n + b_n)$
$b_n - a_n = \frac{1}{2}(b_{n-1} - a_{n-1})$

Figure 1.8

Solution

1. We need to find N such that:

$$\overbrace{|p_N - p| \leq \frac{1}{2^N}(b-a)}^{Error\,bound\,theorem} \leq \texttt{TOL}$$

Then solve for N:

$$\frac{1}{2^N}(b-a) \leq \texttt{TOL} \Longrightarrow 2^{-N} \leq \frac{\texttt{TOL}}{b-a}$$

Hence

$$-N\log_{10}(2) \leq \log_{10}\left(\texttt{TOL}/(b-a)\right) \Longrightarrow N \geq -\log_{10}\left(\texttt{TOL}/(b-a)\right)/\log_{10}(2)$$

and using the ceiling function $\lceil x \rceil$: (least integer greater than or equal to x),

$$N = \left\lceil -\log_{10}\left(\texttt{TOL}/(b-a)\right)/\log_{10}(2)\right\rceil$$

2. For $a = 1$, $b = 2$ and $\texttt{TOL} = 10^{-3}$:

```python
import numpy as np
TOL=10**-3
a, b = 1 ,2
N=np.ceil(- np.log10 ( TOL/(b-a) ) /np.log10(2) )
print(f"we need at least {N} iterations to achieve {TOL} accuracy")
```

```
we need at least 10.0 iterations to achieve 0.001 accuracy
```

We can test the convergence of bisection method via the code shown in Listing 1.3 which is a modified version of Listing 1.1 that can take the exact solution as input whenever that is available. In that case, the function computes and displays the absolute error (see Equation 1.2) of the approximation to the solution $p = \sqrt{2}$. The code also computes the relative error defined by

$$\text{Relative Error}: \ RE_n := \frac{|p_n - p|}{|p|}. \tag{1.3}$$

For reasons that we will discuss below we display $-\log_{10}(RE_n)$

From the output of Listing 1.3 we see that both absolute and relative errors decrease and that the $-\log_{10}(RE_n)$ provides a rough estimate of the number of significant figures that we approximate for the exact solution.

Note that at the beginning of the main code in Listing 1.3 we have included a checkpoint to ensure the code will run only if the function satisfies the $f(a)f(b) < 0$ condition for the bisection method. We have done so via the exception `raise ValueError()` that will display the error that is specified (as a string) inside `()`. You can read more about exceptions here.

! Important 2: Computing Activity (Creating a module)

- The aim of this activity is to use the code from Listing 1.3 as a module that you will call in a test script. Recall that a module can be imported and contains Python definitions and statements. To get started, copy Listing 1.3 an save it into a file name `rootsolvers.py`.
- Copy the code below into a file named `main_bisection.py` and run it. This will import `rootsolvers.py` as module with alias `rs`, so you can access your function via `rs.bisection(f,a,b,Nmax)`.

Listing 1.4 `main_bisection.py`

```python
import numpy as np
import matplotlib.pyplot as plt
import rootsolvers as rs

#function
f = lambda x: x**3+x**2-2*x-2
#interval
a, b = 1.0, 2.0
#maximum number of iterations
Nmax = 5
#exact solution:
pe=np.sqrt(2)
#run Bisection Method:
bisec_approx=rs.bisection(f,a,b,Nmax,pe)
print("Approximations are:\n", bisec_approx)
```

Your output should be:

```
n 1     p_n:1.50000000    E_n:0.08578644    RE_n:0.06066017    -log10(RE_n):1.22
n 2     p_n:1.25000000    E_n:0.16421356    RE_n:0.11611652    -log10(RE_n):0.94
n 3     p_n:1.37500000    E_n:0.03921356    RE_n:0.02772818    -log10(RE_n):1.56
n 4     p_n:1.43750000    E_n:0.02328644    RE_n:0.01646600    -log10(RE_n):1.78
n 5     p_n:1.40625000    E_n:0.00796356    RE_n:0.00563109    -log10(RE_n):2.25
Approximations are:
 [1.5     1.25    1.375   1.4375  1.40625]
```

1.2.4 Other stopping criteria

In practice we do not need to compute the number of iterations N to achieve `TOL` accuracy. Instead we can simply specify a large maximum number of iterations N_{max} and use the error bound $\frac{1}{2}(b_n - a_n) <$ `TOL` to stop the algorithm when the error reaches the desired accuracy. An example is given in Algorithm 1.2

Algorithm 1.2 Bisection with error bound as stopping criteria

```
1: Inputs: $f \in C[a,b]$ with $f(a)f(b) < 0$,
2:     $N_{\max}$: Max no. of iterations.
3:     TOL: error tolerance
4: Output: $p$: approximation to the solution of $f(p) = 0$.
5: procedure BisectionWithErrorBound($f, a, b, N_{max}$,TOL)
6:     Set $n = 1$ and $f_a = f(a)$
7:     while $n \le N_{max}$ do
8:         $p = (a+b)/2$ and $f_p = f(p)$
9:         if $(b-a)/2 <$ TOL then
10:            break
11:            Output ($p$)
12:        end if
13:        Set $n = n + 1$
14:        if $f_a \cdot f_p > 0$ then
15:            set $a = p$ and $f_a = f_p$
16:        else
17:            set $b = p$. ($f_a$ is unchanged);
18:        end if
19:    end while
20: end procedure
```

! Important 3: Computing Activity

Implement the bisection method from Algorithm 1.2 as a function and include it in your module `rootsolvers.py` from Important 2. Your function must have the following structure:

```python
def bisection_with_stopping(f,a,b,Nmax,TOL):

    """
    Bisection Method: Returns a numpy array of the
    sequence of approximations obtained by the bisection method.

    Parameters
    ----------
    f : function
        Input function for which the zero is to be found.
    a : float
        Left side of interval.
    b : float
        Right side of interval.
    Nmax : integer
        Number of iterations to be performed.
    TOL : float
        Tolerance used to stop iterations once the n iteration
        satisfies (b-a)/2^n <= TOL.

    Returns
    -------
    p_array : numpy.ndarray,
            Array containing the sequence of approximations.
            The shape is (k,), with k at most Nmax, and k < Nmax when
            the stopping criterion is met before reaching Nmax.
    """

    # Initialise the array with zeros
    p_array = np.zeros(Nmax)

    # Continue here:...



    return p_array
```

Note that this file contains already an unfinished function with the following signature:

```python
def bisection_with_stopping(f,a,b,Nmax,TOL):
```

This function is to return a `numpy.ndarray` called `p_array`, shape `(k, )`, which is a 1-D array of the approximations p_n ($n = 1, 2, \dots, k$) computed by the bisection method when stopped according to Equation 1.2. The value of `k` is the smallest integer such that the stopping criterion holds, unless `Nmax` iteration have been performed, in which case `k = Nmax`.

The input f can be a lambda function or function defined using `def`, `a` and `b` define the initial interval $[a, b]$ of interest, and `Nmax` is the maximum number of iterations to be performed.

- Complete this function so that it implements the bisection algorithm from Algorithm 1.2 , and provides the output as required above. You can review how to write functions in python here.
- Make sure your code raises an exception e.g. `raise RuntimeError()` if the maximum iterations is reached without achieving the desired level of accuracy `TOL`.
- Modify your file `main_bisection.py` from Important 2 as shown below and run it to test your new function.

Listing 1.5 `main_bisection.py`

```python
import numpy as np
import matplotlib.pyplot as plt
import rootsolvers as rs

# Initialise
f = lambda x: x**3 + x**2 - 2*x - 2
a, b =1.0 , 2.0


######## TEST 1
Nmax = 20
TOL=1e-3
print("TEST 1\n")
# Run bisection
p_test1 = rs.bisection_with_stopping(f,a,b,Nmax,TOL)
# Print output
print("Approximations:\n", p_test1)
print(f"Code converged after {len(p_test1)}")

######## TEST 2
Nmax = 10
TOL=1e-6
print("TEST 2\n")
# Run bisection
p_test2 = rs.bisection_with_stopping(f,a,b,Nmax,TOL)
# Print output
print("Approximations:\n", p_test2)
```

```
TEST 1
Approximations:
 [1.5        1.25       1.375      1.4375     1.40625    1.421875
 1.4140625  1.41796875 1.41601562 1.41503906]
Code converged after 10
TEST 2
RuntimeError: Maximum number of iterations reached without finding root.
NameError: name 'p_test2' is not defined
```

Note 2: Note

There are other stopping criteria that we could specify:

- $|f(p_n)| \leq$ `TOL`. This criteria is reasonable since p_n is expected to converge to the solution of $f(p) = 0$. However, there are cases where $|f(p_n)|$ is small but p_n is not close to p.
- $|p_n - p_{n+1}| \leq$ `TOL`. If p_n converges to the root p, then $|p_n - p_{n+1}|$ will converge to zero. Hence this seems like a reasonable stopping criteria. However, there are sequences that do not converge for which $|p_n - p_{n+1}|$ will converge to zero (see Exercise 1.7).
- $\frac{|p_n - p_{n+1}|}{|p_{n+1}|} \leq$ `TOL`. This is a good stopping criteria as it is an approximation of the relative error defined in Equation 1.3.

Please read Appendix B where we review the concept of significant figure/digits and its connection to the relative error.

1.2.5 Convergence rate

Expression from Equation 1.2 not only tells us how accurate the method is but also how fast it converges to a solution (root of f). Note that

$$E_n = |p_n - p| \leq \frac{b-a}{2^n} = \frac{C}{2^n} \tag{1.4}$$

where we have defined $C = b - a$ which is constant with respect to n. From the definition of Big-O notation (see Definition A.2), the previous expression can be written as

$$|p_n - p| = \mathcal{O}\left(\frac{1}{2^n}\right) \qquad \text{as} \quad n \to \infty$$

which in turn tells us that, for sufficiently large n, the sequence $\{p_n\}_{n=1}^{\infty}$ will go to zero as fast as the sequence $\{2^{-n}\}_{n=1}^{\infty}$.

We now wish to numerically verify the error bound from Equation 1.2. To this end, let us take the $\log_{10}$ from both sides of Equation 1.2:

$$\log_{10} E_n := \log_{10} |p_n - p| \leq \log_{10}\left[\frac{b-a}{2^n}\right] = \log_{10}(b-a) - (\log_{10} 2)n$$

Thus, if we plot $\log_{10} E_n$ versus n, the error bound predicts that the $\log_{10}$ of the error is bounded by a straight line with intercept $\log_{10}(b-a)$ and slope $\log_{10}(2)$. We verify this in the following Important 4 where we used the setting of Exercise 1.1.

! Important 4: Computing Activity (Plotting Errors)

In the file `main_bisection.py` from Important 4 include the code below to plot error convergence for the bisection method. The code uses Object-Oriented (OO) style in `matplotlib` to produce the figures. See this tutorial tutorial.

```
#make sure rootsolvers and numpy are already imported!

import matplotlib.pyplot as plt

# Initialise
f = lambda x: x**3 + x**2 - 2*x - 2

a, b= 1.0, 2.0
Nmax = 100
TOL=1e-8
pe=np.sqrt(2)

# Run bisection
p = rs.bisection_with_stopping(f,a,b,Nmax,TOL)
# Print output
print(f"Bisection converges in {len(p)} iterations")
```

```
Bisection converges in 27 iterations
```

```
error = np.abs(p - pe) # compute error
n = 1 + np.arange(np.shape(p)[0])


fig, ax = plt.subplots()
ax.set_yscale("log")
ax.set_xlabel("Iteration: $n$")
ax.set_title("Convergence behaviour")
ax.grid(True)

# Plot error bound
ax.plot(n , (b-a)/2**n , "o", label="$1/2^n$",linestyle="-")
# Plot error bisection
ax.plot(n , error , "o", label="Error Bisection",linestyle="-")


# Add legend
ax.legend(fontsize=16, loc ='best');
```

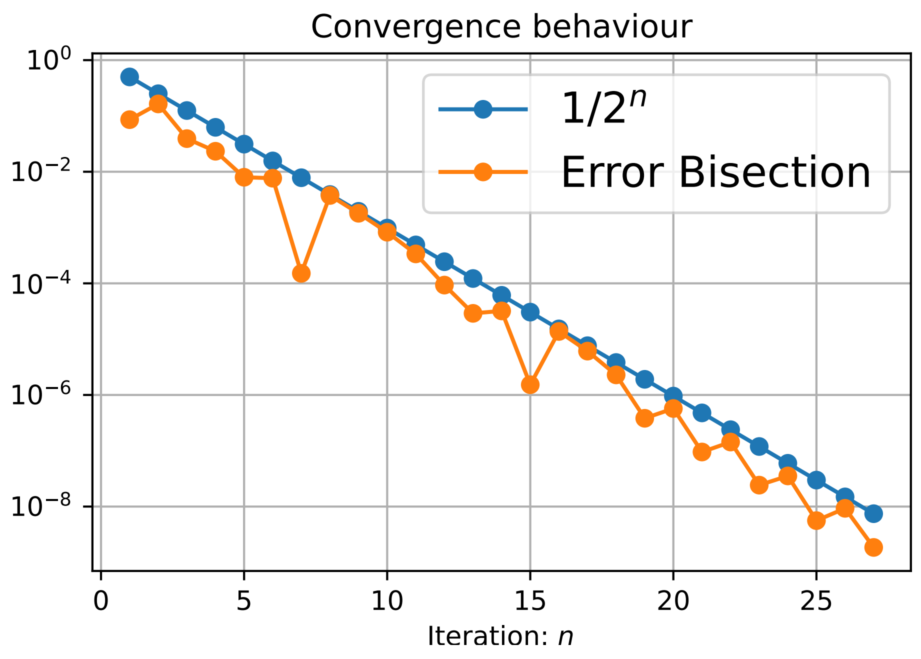


Figure 1.9: Convergence behaviour of the bisection method for the function of Exercise 1.1

Going back to the general case, let us now inspect how the error obtained at iteration $n$ decreases with respect to iteration $n-1$. For sufficiently large $n$, from Equation 1.4 we have that

$$|p_n - p| \approx \frac{C}{2^n}$$

then,

$$\frac{|p_n - p|}{|p_{n-1} - p|} \approx \frac{\frac{C}{2^n}}{\frac{C}{2^{n-1}}} = \frac{1}{2} \Longrightarrow |p_n - p| \approx \frac{1}{2}|p_{n-1} - p| \tag{1.5}$$

Hence the error is cut in half at every iteration (when $n$ is large enough). In general, when there exists λ with $0 < \lambda < 1$ such that error satisfies:

$$\frac{|p_n - p|}{|p_{n-1} - p|} \approx \lambda \tag{1.6}$$

for sufficiently large n we say that $\{p_n\}_{n=1}^{\infty}$ **converges linearly** to p and we call λ the convergence rate. The smaller the λ the faster the convergence. However, note that the error decreases by the same factor (rate) λ at every iteration.

1.2.6 Additional Exercises

Exercise 1.3. Use the bisection method to find p_3 for $f(x) = \sqrt{x} - \cos x = 0$ on $[0, 1]$.

Solution

Final sol: $p_3 = 0.625$. You may want to apply bisection method via hand calculations to achieve this answer to better understand the method. Then, you can use any of the functions that you have implemented in the previous computing activities. For the answers below, I used the function `bisection(f,a,b,Nmax)` which following Important 2 is in the file `rootsolvers.py` which I can import as module and conduct the following computations.

```
import numpy as np
import rootsolvers as rs
#function
f = lambda x: np.sqrt(x)-np.cos(x)
#interval
a, b = 0.0, 1.0
#maximum number of iterations
Nmax = 10

#run Bisection Method:
bisec_approx=rs.bisection(f,a,b,Nmax)
print("Approximations from bisection method:\n", bisec_approx)
#We can check this converges to a root via:
print("f(bisec_approx):\n", f(bisec_approx))
```

```
Approximations from bisection method:
 [0.5        0.75       0.625      0.6875     0.65625    0.640625
 0.6484375  0.64453125 0.64257812 0.64160156]
f(bisec_approx):
 [-1.70475781e-01  1.34336535e-01 -2.03937045e-02  5.63212514e-02
  1.78067276e-02 -1.33182442e-03  8.22774028e-03  3.44554526e-03
  1.05625921e-03 -1.37932657e-04]
```

Exercise 1.4. Consider the bisection method for finding the root of $f(x) = 2(\tan x) - 3x$ that lies between $\frac{\pi}{6}$ and $\frac{\pi}{3}$.

1. Show that there is indeed a root between $\frac{\pi}{6}$ and $\frac{\pi}{3}$.
2. Assuming the interval is $\left(\frac{\pi}{6}, \frac{\pi}{3}\right)$, how many iterations of the bisection method are needed to guarantee that the (absolute) error is less than 10^{-6}

Solution

1. Note that f is continuous on $\left[\frac{\pi}{6}, \frac{\pi}{3}\right]$, and that $f(\pi/6) < 0$ and $f(\pi/3) > 0$ hence from Theorem 1.1 there exists a root in $\left(\frac{\pi}{6}, \frac{\pi}{3}\right)$.
2. You may want to work directly with the error bound to better understand the derivation:

$$\overbrace{|p_N - p| \leq \frac{1}{2^N}(b-a)}^{Error\, bound\, theorem} \leq 10^{-6}$$

Then solve for N:

$$\frac{1}{2^N}(b-a) \leq 10^{-6} \Longrightarrow 2^{-N} \leq \frac{10^{-6}}{b-a} = \frac{10^{-6}}{\pi/6} \quad (\text{note}\, b-a = \pi/6)$$

Thus

$$-N \log_{10} 2 \leq \log_{10} 10^{-6} - \log_{10}(\pi/6) \Longrightarrow N \geq \frac{6 + \log_{10}\pi - \log_{10} 6}{\log_{10} 2} \approx 18.998$$

Thus we need $N \geq 19$ to achieve an accuracy of 10^{-6}.
You can check with the code and the formula derived in Exercise 1.2

```
import numpy as np
TOL=10**-6
a, b = np.pi/6 ,np.pi/3
N=np.ceil(- np.log10 ( TOL/(b-a) ) /np.log10(2) )
print(f"we need at least {N} iterations to achieve {TOL} accuracy")
```

```
we need at least 19.0 iterations to achieve 1e-06 accuracy
```

Exercise 1.5. Use Theorem 1.2 to find a bound for the number of iterations needed to achieve an approximation with accuracy 10^{-4} to the solution of $x^3 - x - 1 = 0$ lying in the interval $[1, 2]$. Find an approximation to the root with this degree of accuracy.

Solution

We can either use direct derivation or

```
import numpy as np
TOL=10**-4
a, b = 1.0 , 2.0
N=np.ceil(- np.log10 ( TOL/(b-a) ) /np.log10(2) )
print(f"we need at least {N} iterations to achieve {TOL} accuracy")
```

```
we need at least 14.0 iterations to achieve 0.0001 accuracy
```

and using Listing 1.3

```
import numpy as np
import rootsolvers as rs
#function
f = lambda x: x**3-x-1
#interval
a, b = 1.0, 2.0
#maximum number of iterations
Nmax = 14

#run Bisection Method:
bisec_approx=rs.bisection(f,a,b,Nmax)
print("Approximations from bisection method:\n", bisec_approx)
#We can check this converges to a root via:
print("f(bisec_approx):\n", f(bisec_approx))
```

```
Approximations from bisection method:
 [1.5        1.25       1.375      1.3125     1.34375    1.328125
 1.3203125  1.32421875 1.32617188 1.32519531 1.32470703 1.32495117
 1.3248291  1.32476807]
f(bisec_approx):
 [ 8.75000000e-01 -2.96875000e-01  2.24609375e-01 -5.15136719e-02
  8.26110840e-02  1.45759583e-02 -1.87106133e-02 -2.12794542e-03
  6.20882958e-03  2.03665067e-03 -4.65948833e-05  9.94790971e-04
  4.74038819e-04  2.13707163e-04]
```

Exercise 1.6. Derive a function f for which the Bisection method converges to a value that is not a zero of f.

Solution

Possible solution: The discontinuous function defined by

$$f(x) = \begin{cases} -1 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

on the interval $[-1, 1]$.
Then $p_1 = 0$, $f(p_1) = -1$, $p_2 = 1/2$, $f(p_2) = 1$, $p_3 = 1/4$, $f(p_3) = 1$, $p_4 = 1/8$, $f(p_4) = 1, \ldots$. Note that, as $n \to \infty$, $p_n \to 0$ while $f(p_n) \to 1$.
We can check the bisection method does not converge to a root

```
import numpy as np
import rootsolvers as rs
#function
f = lambda x: -1.0 if(x <= 0) else 1.0
f_vector = np.vectorize(f)
#interval
a, b = -1.0, 1.0
#maximum number of iterations
Nmax = 10

#run Bisection Method:
bisec_approx=rs.bisection(f,a,b,Nmax)
print("Approximations from bisection method:\n", bisec_approx)
#We can check this does NOT converges to a root via:
print("f(bisec_approx):\n", f_vector(bisec_approx))
```

```
Approximations from bisection method:
 [0.         0.5        0.25       0.125      0.0625     0.03125
 0.015625   0.0078125  0.00390625 0.00195312]
f(bisec_approx):
 [-1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

Exercise 1.7.

1. Let $\{p_n\}_{n=1}^{\infty}$ be a sequence defined by $p_n = \sum_{k=1}^{n} \frac{1}{k}$. Show that $\{p_n\}_{n=1}^{\infty}$ diverges even though $\lim_{n\to\infty}(p_n - p_{n-1}) = 0$.

2. Explain why one must be careful when using $|p_{n+1} - p_n| \leq \epsilon$ as a stopping criterion for general sequences. Does this apply to the bisection method?

Solution

1. Since
$$p_n - p_{n-1} = \sum_{k=1}^{n} \frac{1}{k} - \sum_{k=1}^{n-1} \frac{1}{k} = \frac{1}{n}$$
we have that $\lim_{n\to\infty}(p_n - p_{n-1}) = 0$. However $\lim_{n\to\infty} p_n = \infty$ since $\lim_{n\to\infty} p_n$ this is the divergent harmonic series.
2. The stopping criterion $|p_{n+1} - p_n| \leq \epsilon$ may give the impression of an accurate approximation, while actually the sequence $\{p_n\}_{n=1}^{\infty}$ could be divergent (as for the example above). This concern does not apply to the bisection method, since then it is known that the exact solution p falls within the latest sub-interval. In other words
$$|p - p_{n+1}| \leq |p_{n+1} - p_n| \leq \epsilon,$$
hence the stopping criterion is effective and ensures small absolute error $|p - p_{n+1}|$.

1.2.7 Additional Computing Activities

! Important 5: Computing Activity (More testing)

Add to your file `main_bisection.py` more test cases to make sure your code from Important 3 stops at the right iteration. You can use, some of the Section 1.2.6 as test cases to check your implementations.

1.3 Fixed-point Iteration

Let us consider the following definition of a fixed point of a function:

Definition 1.2. Given $g : [a, b] \to \mathbb{R}$ a point $p \in [a, b]$ that satisfies

$$g(p) = p$$

p is called a *fixed point* of g (Figure 1.10).

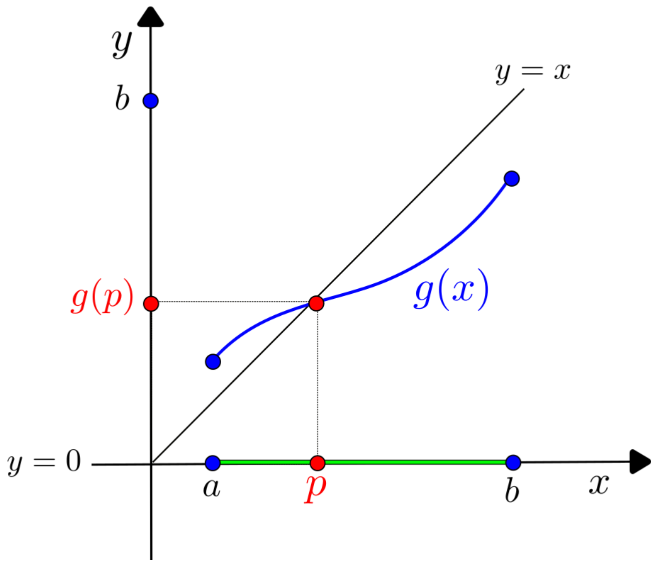


Figure 1.10: Fixed-point

Note that we can write a fixed-point problem as a root-finding problem: Define

$$f(x) = x - g(x)$$

Then

$$f(p) = 0 \Longrightarrow \overbrace{p - g(p)}^{f(p)} = 0 \Longrightarrow g(p) = p$$

i.e. a root of f is a fixed point of g.

We can also write a root-finding problem $f(p) = 0$ as a fixed-point problem. Let us define $g(x) = x - f(x)$. Then

$$g(p) = p \Longrightarrow \overbrace{p - f(p)}^{g(p)} = p \Longrightarrow f(p) = 0$$

In fact we can define $g(x) = x - \lambda f(x)$ for any $\lambda \neq 0$

The aim of this section is to study Fixed-Point Iteration: a method to solve fixed-point problems. However, before we proceed we need to make sure the fixed-point problem at hand has a solution.

Theorem 1.3 (Fixed-point (Existence and Uniqueness))**.**

1. *If $g \in C[a, b]$ 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, $g'(x)$ exists on (a, b) and a positive constant $k < 1$ exists with*

$$|g'(x)| \leq k, \qquad \textit{for all} \;\; x \in (a, b),$$

then there is exactly one fixed point in $[a, b]$.

Proof. The proof requires Intermediate Value Theorem (Theorem A.1) as well as the Mean Value Theorem (Theorem A.2).

1. Assume $g(a) > a$ and $g(b) < b$ (note that if $g(a) = a$ or $g(b) = b$ the proof is trivial). Define $h(x) = g(x) - x$ and note $h \in C[a, b]$. In addition, notice that $h(a) = g(a) - a > 0$ and $h(b) = g(b) - b < 0$. Therefore, by the Intermediate Value Theorem (Theorem A.1), there exists $p \in (a, b)$ such that $h(p) = 0$ and so $g(p) = p$.

2. Prove by contradiction: Assume there are more than one fixed point. In particular, assume that there exists p and q with $q \neq p$ such that satisfy $g(p) = p$ and $g(q) = q$. Note that

$$|p-q| = \overbrace{|g(p)-g(q)| = |g'(\xi)(p-q)|}^{Mean-Value\ Theorem} = \overbrace{|g'(\xi)||p-q| < |p-q|}^{since\ |g'(x)| \leq k < 1}$$

where ξ is between p and q. Therefore $|p - q| < |p - q|$ which is a contradiction!. Hence the fixed point is unique.

□

Exercise 1.8.

1. Let $g(x) = \frac{1}{3}(x^2 - 1)$. Does g have a unique fixed point on the interval $[-1, 1]$?
2. Let $g(x) = e^{-3x}$. Does g have a unique fixed point on the interval $[0, 1]$?

Solution

1.
- g is a polynomial, thus continuous on $[-1, 1]$.
- g maps $[-1, 1]$ onto $[-1/3, 0] \subset [-1, 1]$.
- By Theorem 1.3 (1) a fixed point $p \in [-1, 1]$ exists.
- Note that $g'(x) = \frac{2}{3}x$ and so

$$|g'(x)| \leq \frac{2}{3}|x| < \frac{2}{3}$$

for all $x \in (-1, 1)$, thus by Theorem 1.3 (2) the fixed point is unique.

2.
- g is continuous on $[0, 1]$.
- g maps $[0, 1]$ onto $[e^{-3}, 1] \subset [0, 1]$.
- By Theorem 1.3 (1) a fixed point $p \in [0, 1]$ exists.
- Note that $g'(x) = -3e^{-3x}$ and so

$$|g'(x)| = |-3e^{-3x}| = 3e^{-3x} \in [3e^{-3}, 3]$$

for all $x \in [0, 1]$, thus the second part of Theorem 1.3 cannot be applied to conclude the fixed point is unique. However, the function does have a unique fixed point.

1.3.1 Fixed-Point Iteration — main idea.

Fixed-Point Iteration is a numerical method for finding fixed points. We start with an initial (guess) of the approximation p_0. Then,

Iteration 1: Compute $p_1 = g(p_0)$ Figure 1.11.

Iteration 2: Compute $p_2 = g(p_1)$ Figure 1.12.

Carry on!: $p_3 = g(p_2), \dots, p_n = g(p_{n-1})$ Figure 1.13.

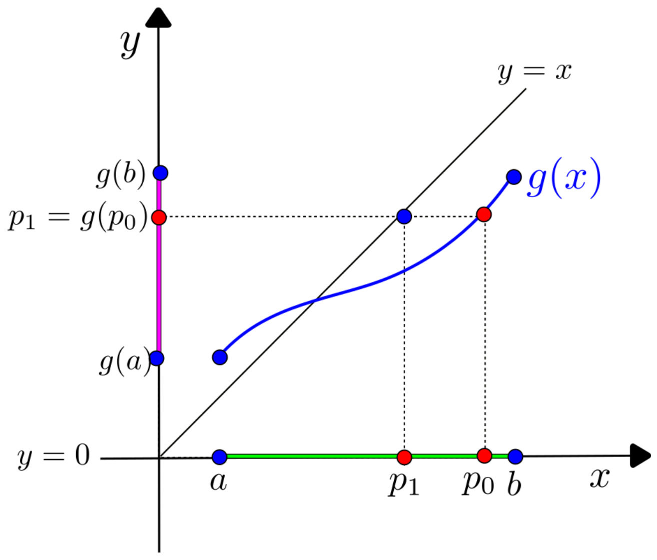


Figure 1.11: First iteration

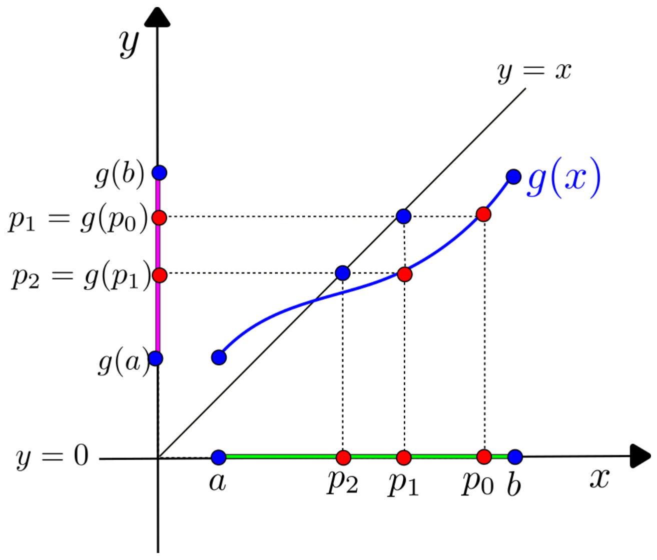


Figure 1.12: Second iteration

Rationale: Suppose the sequence $p_1, p_2 \dots, p_n$ converges to some limit p, i.e. $p_n \to p$ as $n \to \infty$. Then

$$p = \lim_{n\to\infty} p_n = \overbrace{\lim_{n\to\infty} g(p_{n-1}) = g\Big(\lim_{n\to\infty} p_{n-1}\Big)}^{Since\, g\, is\, continuous} = g(p)$$

Hence, p satisfies $p = g(p)$ and thus p is a fixed point.

1.3.2 Pseudocode and Python code

In Algorithm 1.3 we show the pseudocode for Fixed Point Iteration.

A Python implementation of Algorithm 1.3 is shown in Listing 1.6.

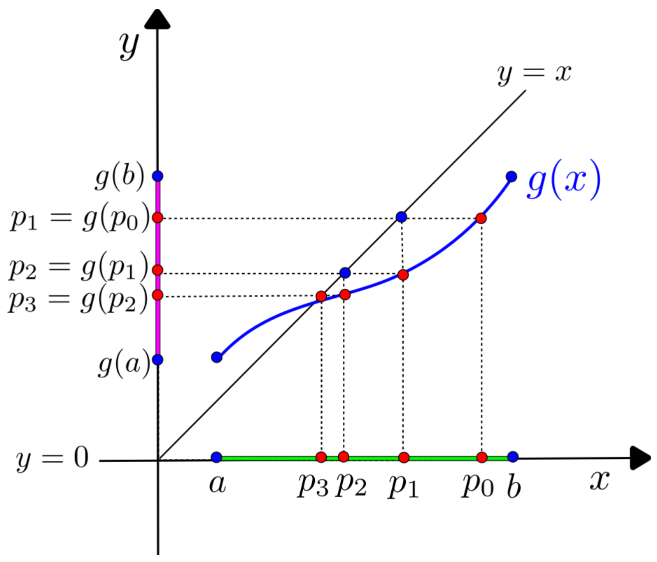


Figure 1.13: Third iteration

! Important 6: Computing Activity

- Add the function `FixedPointIteration` from Listing 1.6 into the `rootsolvers` module that you created for the Computing activities of Section 1.2.
- Save the code below into a file `main_fixed_point.py` to test your python function using $g(x) = \frac{1}{3}(x^2-1)$ on the interval $[-1,1]$ as in Exercise 1.8. Note that the analytic solution to the fixed-point problem (on this interval) is $p = \frac{1}{2}(3-\sqrt{13}) \approx -0.30277563$.

Listing 1.7 `main_fixed_point.py`

```
import numpy as np
import rootsolvers as rs

g = lambda x: 1/3*(x**2 -1)
p0=1.0
Nmax=20
pe=(3-np.sqrt(13))/2
p=rs.FixedPointIteration(g,p0,Nmax)
print(f"Approximation after {Nmax} iterations is {p:5.16f}\n 'Exact'
 ↪  solution: {pe:5.16f}")
```

```
iteration 1      p_n:0.0000000000000000  p_n-g(p_n):0.3333333333333333
iteration 2      p_n:-0.3333333333333333     p_n-g(p_n):-0.0370370370370370
iteration 3      p_n:-0.2962962962962963     p_n-g(p_n):0.0077732053040695
iteration 4      p_n:-0.3040695016003658     p_n-g(p_n):-0.0015555888681974
iteration 5      p_n:-0.3025139127321684     p_n-g(p_n):0.0003145314689896
iteration 6      p_n:-0.3028284442011580     p_n-g(p_n):-0.0000634664069226
iteration 7      p_n:-0.3027649777942354     p_n-g(p_n):0.0000128116128500
iteration 8      p_n:-0.3027777894070854     p_n-g(p_n):-0.0000025859931659
iteration 9      p_n:-0.3027752034139195     p_n-g(p_n):0.0000005219853004
iteration 10     p_n:-0.3027757253992199     p_n-g(p_n):-0.0000001053628945
iteration 11     p_n:-0.3027756200363254     p_n-g(p_n):0.0000000212675475
iteration 12     p_n:-0.3027756413038729     p_n-g(p_n):-0.0000000042928634
iteration 13     p_n:-0.3027756370110095     p_n-g(p_n):0.0000000008665163
iteration 14     p_n:-0.3027756378775258     p_n-g(p_n):-0.0000000001749068
iteration 15     p_n:-0.3027756377026191     p_n-g(p_n):0.0000000000353050
iteration 16     p_n:-0.3027756377379241     p_n-g(p_n):-0.0000000000071264
iteration 17     p_n:-0.3027756377307977     p_n-g(p_n):0.0000000000014385
iteration 18     p_n:-0.3027756377322363     p_n-g(p_n):-0.0000000000002904
iteration 19     p_n:-0.3027756377319458     p_n-g(p_n):0.0000000000000587
iteration 20     p_n:-0.3027756377320045     p_n-g(p_n):-0.0000000000000119
Approximation after 20 iterations is -0.3027756377320045
 'Exact' solution: -0.3027756377319946
```

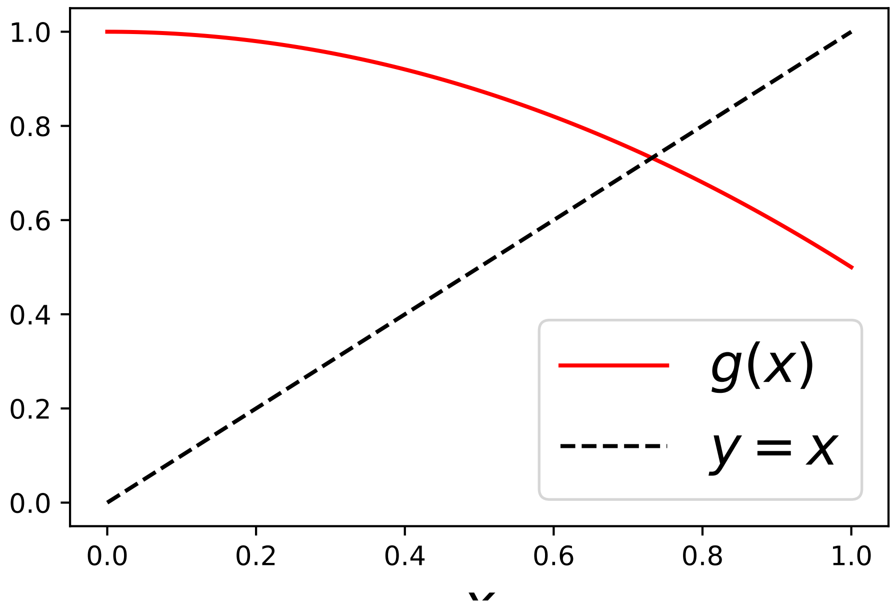
Exercise 1.9.

1. Let $g(x) = 1 - \frac{1}{2}x^2$. Sketch the function $g(x)$ and the function $y = x$ in one figure for $x \in [0, 1]$
2. Use fixed point iteration to numerically solve $x = g(x)$. Use a starting guess $p_0 = 1$, and compute $p_1, p_2, \dots, p_{20}$. Does it converge to the fixed point $p = -1 + \sqrt{3}$?
3. Is it possible to find a function g for which fixed-point iteration does not converge to its fixed point?

Solution

For 1:

```python
import numpy as np
import matplotlib.pyplot as plt
g = lambda x: 1-0.5*x**2
a, b=0, 1.0
x = np.linspace(a,b,100)
# plotting
fig, ax = plt.subplots()
ax.plot(x,g(x), color="red", label="$g(x)$")
ax.plot(x,x, color="black", label="$y=x$",linestyle="--")
ax.set_xlabel("$x$",fontsize=20)
ax.legend(fontsize=20)
```



For 2:
Using Listing 1.6 (making sure `rootsolvers` has been imported first as `rs`):

```python
g = lambda x: 1-0.5*x**2
Nmax=20
p0=1
p=rs.FixedPointIteration(g,p0,Nmax)
```

```
iteration 1      p_n:0.5000000000000000  p_n-g(p_n):-0.3750000000000000
iteration 2      p_n:0.8750000000000000  p_n-g(p_n):0.2578125000000000
iteration 3      p_n:0.6171875000000000  p_n-g(p_n):-0.1923522949218750
iteration 4      p_n:0.8095397949218750  p_n-g(p_n):0.1372171347029507
iteration 5      p_n:0.6723226602189243  p_n-g(p_n):-0.1016684600591502
iteration 6      p_n:0.7739911202780745  p_n-g(p_n):0.0735222474127288
iteration 7      p_n:0.7004688728653456  p_n-g(p_n):-0.0542028062080304
iteration 8      p_n:0.7546716790733761  p_n-g(p_n):0.0394363506710904
iteration 9      p_n:0.7152353284022857  p_n-g(p_n):-0.0289838841003516
iteration 10     p_n:0.7442192125026372  p_n-g(p_n):0.0211503306316601
iteration 11     p_n:0.7230688818709772  p_n-g(p_n):-0.0155168141639502
iteration 12     p_n:0.7385856960349274  p_n-g(p_n):0.0113401112286265
iteration 13     p_n:0.7272455848063009  p_n-g(p_n):-0.0083113448835699
iteration 14     p_n:0.7355569296898707  p_n-g(p_n):0.0060789280972655
iteration 15     p_n:0.7294780015926052  p_n-g(p_n):-0.0044529210036243
iteration 16     p_n:0.7339309225962295  p_n-g(p_n):0.0032582221677059
iteration 17     p_n:0.7306727004285236  p_n-g(p_n):-0.0023860019957209
iteration 18     p_n:0.7330587024242445  p_n-g(p_n):0.0017462330242031
iteration 19     p_n:0.7313124694000415  p_n-g(p_n):-0.0012785666499653
iteration 20     p_n:0.7325910360500067  p_n-g(p_n):0.0009358491004179
```

```python
pe=-1+np.sqrt(3)

print(f"Approximation after {Nmax} iterations is {p:.16f}\n 'Exact'
 ↪  solution: {pe:.16f}")
```

```
Approximation after 20 iterations is 0.7325910360500067
 'Exact' solution: 0.7320508075688772
```

3. Yes, there are many possibilities. For example: $g(x) = x^2$, $p_0 = 2$.

```python
g = lambda x: x**2
p0=2.0  #initial guess
Nmax=5
p=rs.FixedPointIteration(g,p0,Nmax)
```

```
iteration 1      p_n:4.0000000000000000  p_n-g(p_n):-12.0000000000000000
iteration 2     p_n:16.0000000000000000     p_n-g(p_n):-240.0000000000000000
iteration 3    p_n:256.0000000000000000   p_n-g(p_n):-65280.0000000000000000
iteration 4    p_n:65536.0000000000000000  p_n-g(p_n):-4294901760.0000000000000000
iteration 5    p_n:4294967296.0000000000000000    p_n-g(p_n):-18446744069414584320.00000
```

So: $p_n \to \infty$ as $n \to \infty$.

1.3.3 Convergence

When is the fixed-point iteration method guaranteed to converge? How quick is the convergence?

Theorem 1.4 (Fixed-point Theorem). *Let $g \in C[a,b]$ be such that $g(x) \in [a,b]$ for all $x \in [a,b]$. Suppose, in addition that $g'(x)$ existis on (a,b) and that a constant $0 < k < 1$ exists with*

$$|g'(x)| \leq k, \qquad \textit{for all} \;\; x \in (a,b),$$

Then, for any number $p_0 \in [a,b]$, the sequence defined by

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

converges to the unique fixed point $p \in [a,b]$.

Proof. First note from Theorem 1.3 that a unique fixed point $p \in [a,b]$ exists. Notice that each p_n $(n \geq 1)$ is well-defined and is in $[a,b]$ since $p_0 \in [a,b]$ and $g : [a,b] \to [a,b]$.

Let us look at the error:

$$\begin{aligned} |p_n - p| &= \overbrace{|g(p_{n-1}) - g(p)| = |g'(\xi_n)(p_{n-1} - p)|}^{Mean-Value\ Theorem\ (for\ some\ \xi_n\ between\ p_n\ and\ p)} \\ &= \overbrace{|g'(\xi_n)|\,|p_{n-1} - p| \leq k|p_{n-1} - p|}^{assumption:\ |g'(x)|\leq k} \end{aligned}$$

So we can apply the same to get $|p_{n-1} - p| \leq k|p_{n-2} - p|$ and

$$0 \leq |p_n - p| \leq k|p_{n-1} - p| \leq k^2|p_{n-2} - p| \leq \cdots \leq k^n|p_0 - p| \tag{1.7}$$

Therefore

$$0 \leq \lim_{n\to 0} |p_n - p| \leq \overbrace{\lim_{n\to 0} k^n|p_0 - p| = 0}^{since\ 0<k<1}$$

Thus $p_n \to p$ as $n \to \infty$. □

Corollary 1.1. *If g satisfies the hypothesis of Theorem 1.4, then the bound for the error involved in using p_n to approximate p is*

$$|p_n - p| \leq k^n \max\{p_0 - a, b - p_0\}$$

Proof. Notice that $p_0, p \in [a, b]$. Thus, if $p \leq p_0$ then $|p_0 - p| = p_0 - p \leq p_0 - a$. If $p \geq p_0$, then $|p_0 - p| = p - p_0 \leq b - p_0$ and so

$$|p_0 - p| \leq \max\{p_0 - a, b - p_0\} \tag{1.8}$$

On the other hand, from the previous proof we have (see Equation 1.7)

$$|p_n - p| \leq k^n|p_0 - p|$$

which combined with Equation 1.8 completes the proof. □

Caution 1: Caution

Corollary 1.1 can be used (see Exercise 1.10) to determine the number of iterations required to achieve some level of accuracy (denoted by `TOL`), e.g. if we could find N by solving

$$k^n \max\{p_0 - a, b - p_0\} \leq \texttt{TOL}$$

which, of course, requires us to know/compute k (derivative bound). In practice, however, this is not very useful as for some functions we do not have analytic expressions for the derivative. Fortunately, as we discussed in Note 2, we can use for example

$$|p_n - p_{n-1}| \leq \texttt{TOL} \tag{1.9}$$

to stop the fixed point iteration. Note that, for fixed point iteration, since by definition $p_n = g(p_{n-1})$ the criteria from Equation 1.9 becomes

$$|g(p_{n-1}) - p_{n-1}| \leq \texttt{TOL}$$

which, we expect to be small whenever p_n is close to the fixed point of g.

In Algorithm 1.4 we show the pseudocode for Fixed Point Iteration with the stopping criteria from Equation 1.9.

! Important 7: Computing Activity

In this activity you will implement fixed-point iteration with stopping criteria as shown in Algorithm 1.4 . To this end, complete the function from Listing 1.8 and include this function in the module `rootsolvers.py` that you wrote for the Computing activities from Section 1.2. This function should return a `numpy.ndarray` called `p_array`, with shape `(k, )`, which is a 1-D array of the approximations p_n $(n = 1, 2, \ldots, k)$ computed by the fixed-point iteration method with the above stopping criterion $|p_n - p_{n-1}| \leq \texttt{TOL}$.
The value of `k` is the smallest integer such that the stopping criterion holds, unless `Nmax` iteration have been performed, in which case `k = Nmax`.

- Complete this function so that it implements the output as required above.
- Make sure your code raises an exception e.g. `raise RuntimeError()` if the maximum iterations is reached without achieving the desired level of accuracy `TOL`.

Listing 1.8 Implementation of fixed-point iteration with stopping criteria

```python
def FixedPointIteration_with_stopping(g,p0,Nmax, TOL):
    """
    Fixed-point iteration  method with stopping criterion:
    Returns a numpy array of the sequence of approximations
    obtained by the fixed point iteration method
    using an early stopping criterion.


    Parameters
    ----------
    g : function
        Input function for which the fixed point is to be found.
    p0 : float
        Initial approximation of fixed point
    Nmax : integer
        Number of iterations to be performed
    TOL : float
        Tolerance used to stop iterations, once |p_k-p_{k-1}| <= TOL.

    Returns
    -------
    p_array : numpy.ndarray
        Array containing the sequence of approximations.
        The shape is (k,), with k at most Nmax, and k < Nmax when
        the stopping criterion is met before reaching Nmax.

    """

    # Initialise the array with zeros
    p_array = np.zeros(Nmax)

    # Continue here:...

    return p_array
```

Modify the code `main_fixed_point.py` from Important 6 to test your new function.

Listing 1.9 main_fixed_point.py

```python
import numpy as np
import rootsolvers as rs

g = lambda x: 1/3*(x**2 -1)
p0=1.0

######## TEST 1
Nmax=10
TOL=1e-4
print("TEST 1\n")
p_test1=rs.FixedPointIteration_with_stopping(g,p0,Nmax,TOL)
print("Approximations:\n",   p_test1)
print(f"Code converged after {len(p_test1)}")


######## TEST 2
Nmax=10
TOL=1e-10
print("TEST 2\n")
p_test2=rs.FixedPointIteration_with_stopping(g,p0,Nmax,TOL)
print("Approximations:\n",   p_test2)
print(f"Code converged after {len(p_test2)}")
```

```
TEST 1
Approximations:
 [ 0.         -0.33333333 -0.2962963  -0.3040695  -0.30251391 -0.30282844
 -0.30276498]
Code converged after 7
TEST 2
RuntimeError: Maximum number of iterations reached without finding root.
NameError: name 'p_test2' is not defined
NameError: name 'p_test2' is not defined
```

1.3.4 Convergence rate

From Corollary 1.1 we have that error is

$$|p_n - p| \leq Ck^n \Longrightarrow |p_n - p| = \mathcal{O}(k^n) \tag{1.10}$$

where we have defined $C = \max\{p_0 - a, b - p_0\}$.

Following the same arguments from Section 1.2.5, if we assume that $|p_n - p| \approx Ck^n$ for sufficiently large n, then

$$\frac{|p_n - p|}{|p_{n-1} - p|} \approx \frac{Ck^n}{Ck^{n-1}} = k.$$

Since $k \in (0,1)$, this expression means that $\{p_n\}_{n=1}^{\infty}$ converges linearly to p with convergence rate k which we recall is the bound for $|g'(x)|$. The lower the k, the faster the convergence of the fixed point iteration.

In addition, from Equation 1.10 we have that the error satisfies

$$\log_{10}|p_n - p| \leq \log_{10} Ck^n = \log_{10} C + (\log_{10} k)n.$$

Therefore, on a semilog plot, the error of the approximation is bounded by a straight line with slope $(\log_{10} k)$. Since $k \in (0,1)$, the slope is negative and will be larger the smaller the k.

Let us verify this convergence behaviour for the function $g(x) = \frac{1}{3}(x^2 - 1)$ on the interval $[-1,1]$. In Exercise 1.8 we computed the value $k = \frac{2}{3}$. If we select $p_0 = 1$ as we did for the code in Listing 1.6, we have $C = \max\{p_0 - a, b - p_0\} = \max\{1 - (-1), 1 - 1\} = 2$. Thus we expect the error to satisfy:

$$|p_n - p| \leq 2\frac{2}{3}^n,$$

which we verify in Important 8.

Important 8: Computing Acivity. Plotting Errors

Add the code below to the end of main_fixed_point.py from Important 7 to plot errors and to compare them with those obtained with the bisection method.

```
import matplotlib.pyplot as plt

# We use the same function as above

Nmax=50
TOL=1e-11




# Run fixed point
p_fp = rs.FixedPointIteration_with_stopping(g,p0,Nmax,TOL)
print(f"Fixed point converged after {len(  p_fp)}")
```

```
Fixed point converged after 17
```

```
#analytical solution:
pe=(3-np.sqrt(13))/2

# Compute errors for fixed point
e_fp = np.abs(pe - p_fp)

n_fp = 1 + np.arange(np.shape(p_fp)[0])

# define function to compare with bisection method
def f(p):
  return g(p)-p

#interval:
a=-1.0
b=1.0

# Run bisection for f(p)=g(p)-p
p_bi = rs.bisection_with_stopping(f,a,b,Nmax,TOL)
print(f"bisection converged after {len(  p_bi)}")
```

```
bisection converged after 38
```

```
# Compute error for bisection
e_bi = np.abs(pe - p_bi)
n_bi = 1 + np.arange(np.shape(p_bi)[0])

fig, ax = plt.subplots()
ax.set_yscale("log")
ax.set_xlabel("$n$")
ax.set_title("Convergence behaviour")
ax.grid(True)

# Plot error bisection
ax.plot(n_bi , e_bi , "o", label="Error Bisection",linestyle="-")
# Plot error bound bisection
ax.plot(n_bi , 2/2**n_bi , "o", label="$2/2^n$",linestyle="--")
# Plot error fixed point iteration
ax.plot(n_fp , e_fp , "d", label="Error Fixed Point",linestyle="-")
# Plot error bound fixed point iteration
ax.plot(n_fp , 2*(2/3)**n_fp , "+", label="$2(2/3)^n$",linestyle="--")


# Add legend
ax.legend(fontsize=16, loc ='best');
```

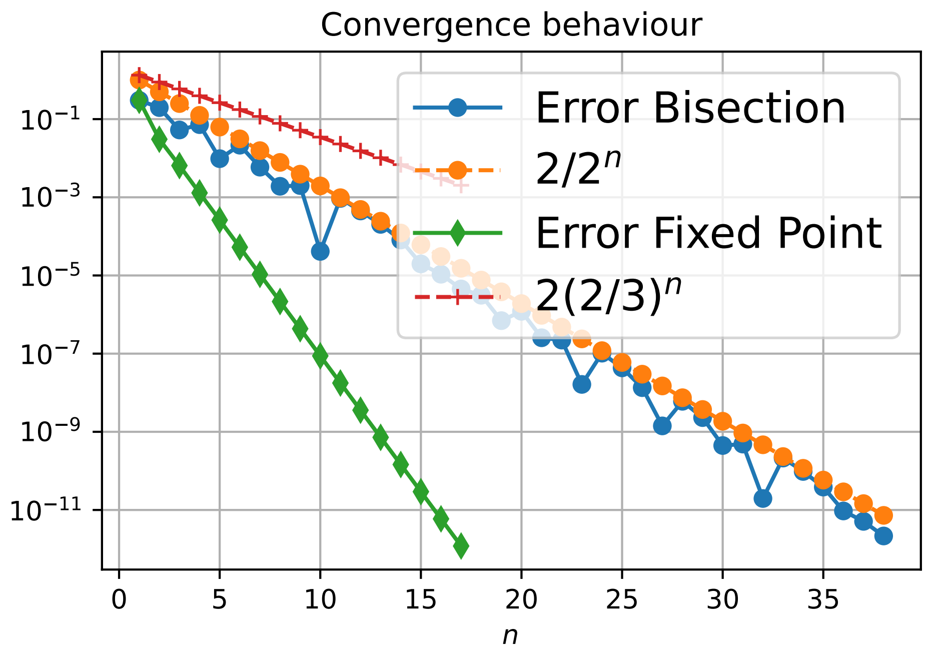


Figure 1.14: Convergence behaviour of fixed point iteration (and bisection method)

From Figure 1.14 we notice that while the error of the fixed-point approximation is indeed bounded by $2\frac{2}{3}^n$, it decreases faster than this error bound. This comes from the fact the error estimate provided by Corollary 1.1 gives us the worst-case scenario. However, nothing prevents the error to approach zero faster than the error bound. For this reason, a stopping criteria based on the error bound may not be optimal. Other stopping criteria was discussed in Caution 1.

In Figure 1.14 we also show the error obtained if we solve the fixed-point problem via the bisection method using the function $f(x) = g(x) - x$. We know from the previous chapter that the error in this case is bounded by $(b-a)/2^n$ which for this case becomes $2(1/2)^n$. We can see that for this function, fixed point iteration reached a level of accuracy 10^{-11} in 17 iterations while the bisection method takes 38 iterations. However, both methods converge linearly to the same value.

1.3.5 Additional Exercises

Exercise 1.10.

1. Use Theorem 1.3 to show that $g(x) = \pi + \frac{1}{2}\sin(x/2)$ has a unique fixed point on $[0, 2\pi]$.

2. Next, use Corollary 1.1, with $p_0 = \pi$, to estimate the number of fixed-point iterations required to achieve 10^{-2} accuracy.

3. Next, use fixed-point iteration to obtain this approximation.

Solution

1. Note that $g(x)$ is continuous. In addition,

$$0 \leq \pi - \frac{1}{2} \leq \overbrace{\pi + \frac{1}{2}\sin(x/2)}^{g(x)} \leq \pi + \frac{1}{2} \leq 2\pi$$

so $g(x) \in [0, 2\pi]$ for all $x \in [0, 2\pi]$. Furthermore,

$$g'(x) = \frac{1}{4}\cos(x/2) \implies |g'(x)| = \left|\frac{1}{4}\cos(x/2)\right| \leq \frac{1}{4} < 1$$

so we can use Theorem 1.3 note ($k = \frac{1}{4}$) to conclude g has a unique fixed point in $[0, 2\pi]$.

2. From Corollary 1.1 we have

$$|p_n - p| \leq k^n \max\{p_0 - a, b - p_0\} = k^n \max\{\pi - 0, 2\pi - \pi\} = k^n\pi = \frac{\pi}{4^n}$$

since $k = \frac{1}{4}$.
So we need to find N such that

$$|p_N - p| \leq \frac{1}{4^N}\pi \leq 10^{-2} \implies -N\log_{10} 4 + \log_{10}\pi \leq -2$$

Thus

$$N\log_{10} 4 - \log_{10}\pi \geq 2 \implies N \geq \frac{2 + \log_{10}\pi}{\log_{10} 4} \approx 4.1477$$

hence we need at least $N = 5$ iterations to achieve 10^{-2} accuracy

3. Assuming you have imported `rootsolvers` as `rs` (and `numpy` as `np`)

```
g = lambda x: np.pi +0.5* np.sin(0.5*x)
Nmax=5
p0=np.pi
p=rs.FixedPointIteration(g,p0,Nmax)
```

```
iteration 1      p_n:3.6415926535897931  p_n-g(p_n):0.0155437891446777
iteration 2      p_n:3.6260488644451154  p_n-g(p_n):-0.0009467579936198
iteration 3      p_n:3.6269956224387352  p_n-g(p_n):0.0000568282133182
iteration 4      p_n:3.6269387942254170  p_n-g(p_n):-0.0000034141256777
iteration 5      p_n:3.6269422083510947  p_n-g(p_n):0.0000002051027956
```

$p_5 \approx 3.626942$

Exercise 1.11. Let A be a given positive constant and $g(x) = 2x - Ax^2$

1. Show that if fixed-point iteration converges to a nonzero limit, then the limit is $p = 1/A$, so the inverse of a number can be found using only multiplications and subtractions.

2. Find an interval about $1/A$ for which fixed-point iteration converges provided p_0 is in that interval.

Solution

1. The non-zero limit, if exists, satisfies

$$p = g(p) \implies p = 2p - Ap^2 \implies 0 = p - Ap^2 \implies 0 = 1 - Ap$$

and thus $p = 1/A$.

2. Note that $g'(x) = 2 - 2Ax$ hence,

$$|g'(x)| \leq 2|1 - Ax|$$

we need $|g'(x)| \leq k$ for some $k < 1$ in some interval. This can be found by

$$|g'(x)| \leq 2|1 - Ax| < 1 \implies |1 - Ax| < \frac{1}{2} \implies -\frac{1}{2} < 1 - Ax < \frac{1}{2}$$

Thus

$$-\frac{1}{2} < 1 - Ax < \frac{1}{2} \implies -\frac{1}{2A} < \frac{1}{A} - x < \frac{1}{2A} \implies \left|\frac{1}{A} - x\right| < \frac{1}{2A}$$

since $A > 0$. This if $x \in \left(\frac{1}{2A}, \frac{3}{2A}\right)$ we have $|g'(x)| = k < 1$.

Let us consider the interval $\left[\frac{1}{2A}, \frac{3}{2A}\right]$. Thus $x \in \left[\frac{1}{2A}, \frac{3}{2A}\right]$ satisfies

$$\left|\frac{1}{A} - x\right| \leq \frac{1}{2A} \implies \left|\frac{1}{A} - x\right|^2 \leq \frac{1}{4A^2} \tag{1.11}$$

On the other hand, $g(x) = 2x - Ax^2$ is continuous and we can write it as

$$g(x) = 2x - Ax^2 = -A\left(x^2 - \frac{2x}{A} + \frac{1}{A^2}\right) + \frac{1}{A} = -A\left(x - \frac{1}{A}\right)^2 + \frac{1}{A}$$

Thus, from Equation 1.11 we have

$$\frac{3}{4A} = -A\frac{1}{4A^2} + \frac{1}{A} \leq \overbrace{-A\left(x - \frac{1}{A}\right)^2 + \frac{1}{A}}^{g(x)} \leq \frac{1}{A}$$

This $g(x) \in \left[\frac{3}{4A}, \frac{1}{A}\right] \subset \left[\frac{1}{2A}, \frac{3}{2A}\right]$ and so Theorem 1.4 we can guarantee convergence of fixed point iteration.

Exercise 1.12. Show that Theorem 1.4 may not hold if inequality $|g'(x)| \leq k$ is replaced by $g'(x) \leq k$. [Hint: Show that $g(x) = 1 - x^2$, for $x \in [0, 1]$ provides a counter example.]

Solution

For $g(x) = 1 - x^2$ we have $g'(x) = -2x \leq 1$ for all $x \in [0, 1]$. However, if $p_0 = 1 \in [0, 1]$, then $p_1 = g(p_0) = g(1) = 0$, $p_2 = g(0) = 1$, $p_3 = g(1) = 0, \ldots$ which does not converge.

1.4 Newton's method

Let us go back to the root finding problem from Definition 1.1. In this section we introduce Newton's which has the advantage that, under some conditions, this methods can achieve faster than linear convergence.

1.4.1 Newton's method main idea

Iteration 1. Let p_0 be an initial (guess) approximation of the root of f. Consider the line tangent to f at the point $(p_0, f(p_0))$:

$$L_0(x) = f(p_0) + f'(p_0)(x - p_0)$$

Recall that $L_0(x)$ is the best linear approximation to $f(x)$ at $x = p_0$. It is then natural to consider the root of L_0 to approximate the root of f. That, is we define p_1 by (see Figure 1.15)

$$L_0(p_1) = 0 \Longrightarrow f(p_0) + f'(p_0)(p_1 - p_0) = 0 \Longrightarrow p_1 = p_0 - \frac{f(p_0)}{f'(p_0)}$$

If p_0 is close to p (the root of f), we expect p_1 to be (perhaps only slightly) closer to p.

Iteration 2. Repeat the procedure and define p_2 to be the root of $L_1(x) = f(p_1) + f'(p_1)(x - p_1)$, i.e. (Figure 1.16)

$$L_1(p_2) = 0 \Longrightarrow p_2 = p_1 - \frac{f(p_1)}{f'(p_1)}$$

Carry on!: We can continue the process (see Figure 1.17) so that the general term is

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \qquad n = 1, 2, \ldots \tag{1.12}$$

An alternative formulation of Newton's method based on Taylor's Theorem can be found in Exercise 1.17.

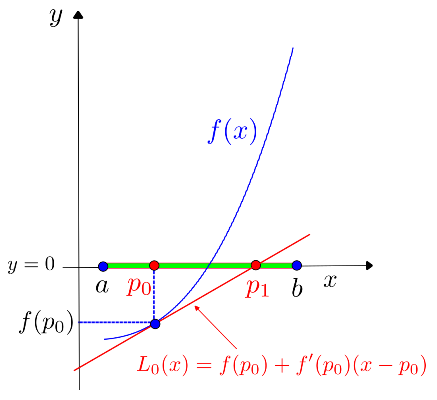


Figure 1.15: First iteration of Newton's method

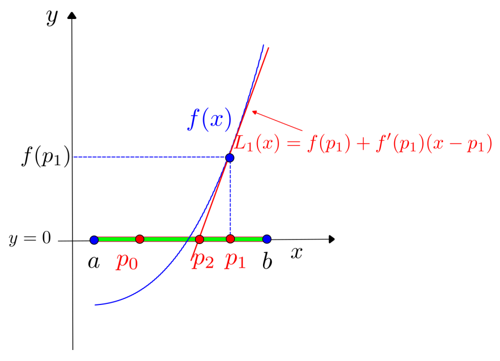


Figure 1.16: Second iteration of Newton's method

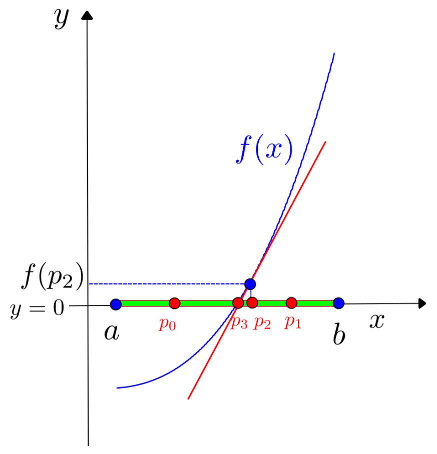


Figure 1.17: Third iteration of Newton's method

Note

In contrast to the bisection method and fixed-point iteration, Newton's method requires knowledge of the derivative of f. Moreover, for Equation 1.12 to be well-defined we need $f'(p_n) \neq 0$ for all $n = 0, 1, \ldots$.

Exercise 1.13. Consider the positive root of $f(x) = x^2 - 2$ (i.e. $p = \sqrt{2}$)

Use Newton's method with $p_0 = 1$ to find the approximation p_1 and p_2. Does it converge?

Solution

```
#function
f = lambda x: x**2-2
#derivative of the function
df= lambda x: 2*x
#initial approximation
p0=1.0
#update to obtain p1
p1=p0-f(p0)/df(p0)
print(f"p1 is {p1:.10f}")
```

```
p1 is 1.5000000000
```

```
#update to obtain p2
p2=p1-f(p1)/df(p1)
print(f"p2 is {p2:.10f}")
```

```
p2 is 1.4166666667
```

The pseudocode for Newton's method is shown in Algorithm 1.5 and a simple Python implementation of Newton's method is shown in Listing 1.10

Important 10: Computing Activity

- Add the function `Newton_simple` from Listing 1.10 into the `rootsolvers` module that you created for the Computing Activities of Section 1.2.
- Save the code below into a file `main_newton.py` to test your python function using $f(x) = x^2 - 2$ with an initial guess $p_0 = 1$ as in Exercise 1.13.
- Try the code with $p_0 = -1$ and then $p_0 = 0$

Listing 1.11 main_newton.py

```python
import numpy as np
import rootsolvers as rs

#function
f = lambda x: x**2-2
#derivative of the function
df= lambda x: 2*x
Nmax=5

np.set_printoptions(precision=16) #to make sure when we print array we have
 ↪  30 decimal places
print("TEST 1\n")
p0=1.0 #initial approximation
p_test1=rs.Newton_simple(f,df,p0,Nmax)
print(f"Approximations from Newton's method starting with p0= {p0}:\n
 ↪  {p_test1}")
print(f"The positive root is {np.sqrt(2):.16f}")

print("TEST 2\n")
p0=-1.0 #initial approximation
p_test2=rs.Newton_simple(f,df,p0,Nmax)
print(f"Approximations from Newton's method starting with p0= {p0}:\n
 ↪  {p_test2}")
print(f"The negative root is {-np.sqrt(2):.16f}")

print("TEST 3\n")
p0=0 #initial approximation
p_test3=rs.Newton_simple(f,df,p0,Nmax)
print(f"Approximations from Newton's method starting with p0= {p0}:\n
 ↪  {p_test3}")
```

```
TEST 1
Approximations from Newton's method starting with p0= 1.0:
 [1.5                1.4166666666666667 1.4142156862745099
 1.4142135623746899 1.4142135623730951]
The positive root is 1.4142135623730951
TEST 2
Approximations from Newton's method starting with p0= -1.0:
 [-1.5                -1.4166666666666667 -1.4142156862745099
 -1.4142135623746899 -1.4142135623730951]
The negative root is -1.4142135623730951
TEST 3
ZeroDivisionError: division by zero
NameError: name 'p_test3' is not defined
```

1.4.2 Convergence

Remark 1.1. Newton method can be written as a fixed point iteration. Consider

$$g(x) = x - \frac{f(x)}{f'(x)} \tag{1.13}$$

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

Note

We introduce the space of twice continuously differentiable functions (see Definition A.4):

$$C^2[a,b] = \left\{ f : [a,b] \to \mathbb{R} \middle| \ f, \ f' \text{ and } f'' \text{ are continuous in } [a,b] \right\}$$

Convergence of Newton's method is guaranteed by the following theorem.

Theorem 1.5 (Convergence of Newton's method)**.** *Let $f \in C^2[a,b]$. If $p \in (a,b)$ is such that $f(p) = 0$ and $f'(p) \neq 0$, then there exists a $\delta > 0$ such that Newton's method generates a sequence $\{p_n\}_{n=1}^{\infty}$ converging to p for any initial approximation $p_0 \in [p-\delta, p+\delta]$.*

Proof. As discussed in Remark 1.1 we have that Newton's method can be written as a fixed-point iteration with g defined by Equation 1.13. Let us show that g satisfies the assumptions of Theorem 1.4 on an interval $[p-\delta, p+\delta]$.

1. We first show that g defined in Equation 1.13 is well-defined on some interval that contains p. Since $f'(p) \neq 0$, then from the continuity of f' there is δ_1 such that $f'(x) \neq 0$ for all $x \in [p-\delta_1, p+\delta_1]$. This means that g is well defined on $[p-\delta_1, p+\delta_1]$

2. Let us now show that there exists $\delta > 0$ and $0 < k < 1$ so that

$$|g'(x)| \leq k \tag{1.14}$$

for all $x \in [p-\delta, p+\delta]$. To this end, note that

$$g'(x) = \left[x - \frac{f(x)}{f'(x)} \right]' = \frac{f(x)f''(x)}{(f'(x))^2} \tag{1.15}$$

where we have used the fact that f'' exists (since $f \in C^2[a,b]$).

Since $f'(p) \neq 0$ and $f(p) = 0$ then

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

Using again the fact that $f \in C^2[a,b]$, we have that g' is continuous. Let us use the continuity of g' at $x = p$: for any $k < 1$ arbitrary, there is $\delta_2 > 0$ such that $|g'(x)| \leq k$ whenever $|x-p| \leq \delta_2$ (or $x \in [p-\delta_2, p+\delta_2]$) which shows Equation 1.14. Take $\delta = \min\{\delta_1, \delta_2\}$ to ensure that g is well-defined and $|g'(x)| \leq k$ for all $x \in [p-\delta, p+\delta]$.

3. We now show that g maps $[p-\delta, p+\delta]$ to $[p-\delta, p+\delta]$. Let $x \in [p-\delta, p+\delta]$, then using that $p = g(p)$, the Mean Value Theorem Theorem A.2 and that $|g'(x)| \leq k < 1$ for all $x \in [p-\delta, p+\delta]$ we have

$$|g(x) - p| = \overbrace{|g(x) - g(p)| = |g'(\xi)||x-p|}^{\textit{Mean Value Theorem } (\xi \textit{ between } g(p) \textit{ and } g(x))} \leq k|x-p| < |x-p| \leq \delta$$

We have thus shown that all the assumptions of Theorem 1.3 are satisfied and hence the fixed-point iteration

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

converges to a unique fixed point $p \in [p-\delta, p+\delta]$. Thus, the Newton method converges to the fixed point of $g(x)$, i.e.

$$g(p) = p - \frac{f(p)}{f'(p)} = p \Longrightarrow -\frac{f(p)}{f'(p)} = 0 \Longrightarrow f(p) = 0$$

since $f'(p) \neq 0$. Thus Newton's method converges to the root p of f provided $p_0 \in [p-\delta, p+\delta]$. □

! Important 10: Computing Activity

In Note 2 we discussed stopping criteria that can be used to stop Bisection method given some tolerance `TOL`. We can also used those stopping rules for other iterative methods such as the Newton method. Have a go at modifying `Newton_simple` from Listing 1.10 so that it stops using some of the criteria from Note 2. You may want to try, for example, $|f(p_n)| \leq$ `TOL`

1.5 The secant method

Often the derivative of f is not available. In this case, the secant method can be used to achieve convergence that can be faster than the bisection method.

1.5.1 Secant method main idea

We start with two initial approximations p_0 and p_1 and consider the secant line through $(p_0, f(p_0))$ and $(p_1, f(p_1))$:

$$S_{01}(x) = f(p_1) + \frac{f(p_1) - f(p_0)}{p_1 - p_0}(x - p_1)$$

The approximation p_2 is computed as the root of S_{01}, i.e.

$$S_{01}(p_2) = 0 \Longrightarrow p_2 = p_1 - f(p_1)\frac{p_1 - p_0}{f(p_1) - f(p_0)}$$

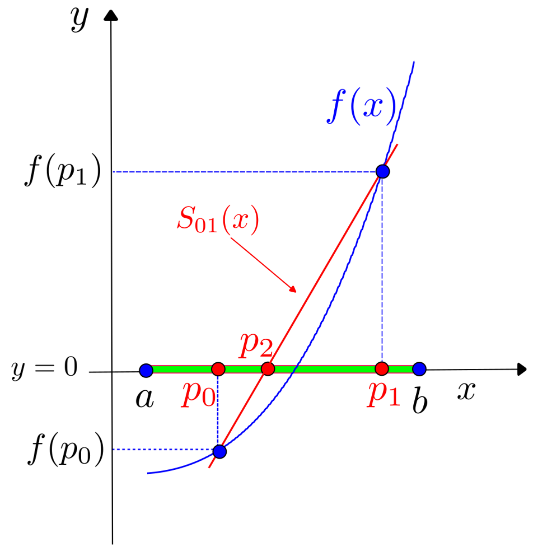


Figure 1.18: Secant method (1st iteration).

we continue the process:

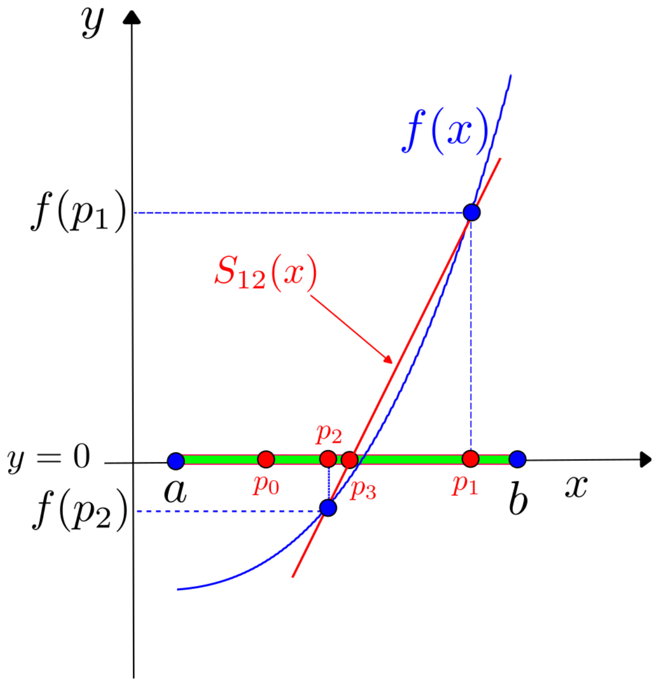


Figure 1.19: Secant method (2nd iteration).

In general, given p_{n-2} and p_{n-1}, define p_n as the root of the secant line through $(p_{n-2}, f(p_{n-2})$ and $(p_{n-1}, f(p_{n-1})$

$$p_n = p_{n-1} - f(p_{n-1})\frac{p_{n-1} - p_{n-2}}{f(p_{n-1}) - f(p_{n-2})} \tag{1.16}$$

Exercise 1.14. Consider the positive root of $f(x) = x^2 - 2$ (i.e. $p = \sqrt{2}$)

- Use the secant method with $p_0 = 2$ and $p_1 = 1$ to find the approximation p_2 and p_3. Does it converge?
- Are the secant approximations the same if p_0 and p_1 are swapped? (i.e. $p_0 = 1$ and $p_1 = 2$)

Solution

```
#function
f = lambda x: x**2-2
#derivative of the function
#initial approximations
p0, p1=2.0, 1.0
#update to obtain p2
p2 =p1 - f(p1) * (p1 - p0) /(f(p1)-f(p0))
print(f"p2 is {p2:.10f}")
```

```
p2 is 1.3333333333
```

```
#update to obtain p3
p3 =p2 - f(p2) * (p2 - p1) /(f(p2)-f(p1))
print(f"p3 is {p3:.10f}")
```

```
p3 is 1.4285714286
```

```
#update to obtain p4
p4 =p3 - f(p3) * (p3 - p2) /(f(p3)-f(p2))
print(f"p4 is {p4:.10f}")
```

```
p4 is 1.4137931034
```

Let's swap p_0 and p_1:

```
#function
f = lambda x: x**2-2
#derivative of the function
#initial approximations
p0, p1=1.0, 2.0
#update to obtain p2
p2 =p1 - f(p1) * (p1 - p0) /(f(p1)-f(p0))
print(f"p2 is {p2:.10f}")
```

```
p2 is 1.3333333333
```

```
#update to obtain p3
p3 =p2 - f(p2) * (p2 - p1) /(f(p2)-f(p1))
print(f"p3 is {p3:.10f}")
```

```
p3 is 1.4000000000
```

```
#update to obtain p4
p4 =p3 - f(p3) * (p3 - p2) /(f(p3)-f(p2))
print(f"p4 is {p4:.10f}")
```

```
p4 is 1.4146341463
```

thus the sequence of iterations are different depending of order of the two initial approximations although they both see to coneverge to the solution

```
print(f"{np.sqrt{2}:.16f}")
```

The pseudocode for the Secant method is shown in Algorithm 1.6 .

Caution 2: Caution

When we iterate the secant method for long enough we, of course, expect that $f(p_n) \approx 0$ but also $f(p_{n-1}) - f(p_n) \approx 0$ hence the denominator in Equation 1.16 will be small. In terms of the Algorithm 1.6 , we need to make sure that if the denominator $q_1 - q_0$ of the update formula is close to zero to machine precision, we replace the denominator by a small number e.g.

$$q_1 - q_0 = \begin{cases} \epsilon & \text{if} \quad |q_1 - q_0| < \epsilon \\ q_1 - q_0 & \text{otherwise} \end{cases}$$

with $\epsilon = 10^{-15}$. Another approach is to simply modify the update formula via:

$$p = p_1 - q_1 \frac{(p_1 - p_0)}{(q_1 - q_0 + \epsilon)}$$

Note

Similar to Newton's method, convergence of the Secant method can be proven when $f'(p) \neq 0$ and when both initial approximations p_0 and p_1 are sufficiently close to the root of f. However, the proof will not be given here but some elements of the convergence behaviour of the Secant method will be studied in Exercise 1.20.

Recall that the bisection method exhibit linear convergence i.e. where the error decreases by a fixed amount (or rate) at each iteration. The advantage of Newton's method and the secant method is that they converge faster than linearly. We will formalise this "faster than linearly" behaviour in the next section, but to motivate the idea in Figure 1.20 we show the plots of the errors (using a $\log_{10}$ scale on the y-axis) that we obtain when we apply Newton's and secant methods to find the root of

$$f(x) = x - \cos(x)$$

which is approximately $p = 0.73908513321516064165531207 04$. We have used $p_0 = 0$ for Newton's method while $p_0 = 0$ and $p_1 = 2$ for the Secant method. We also compare the convergence of these methods with the one obtained via the bisection method on the interval $[0, 2]$ and the fixed point iteration with the function $g(x) = \cos(x)$ and initial guess $p_0 = 1$. Recall that $g(p) = p$ implies $0 = p - g(p) = f(p)$. As we expect, fixed-point iteration and bisection method produce approximations that converge linearly to the root of f (or fixed point of g) while Newton's and secant method converge "faster than linearly".

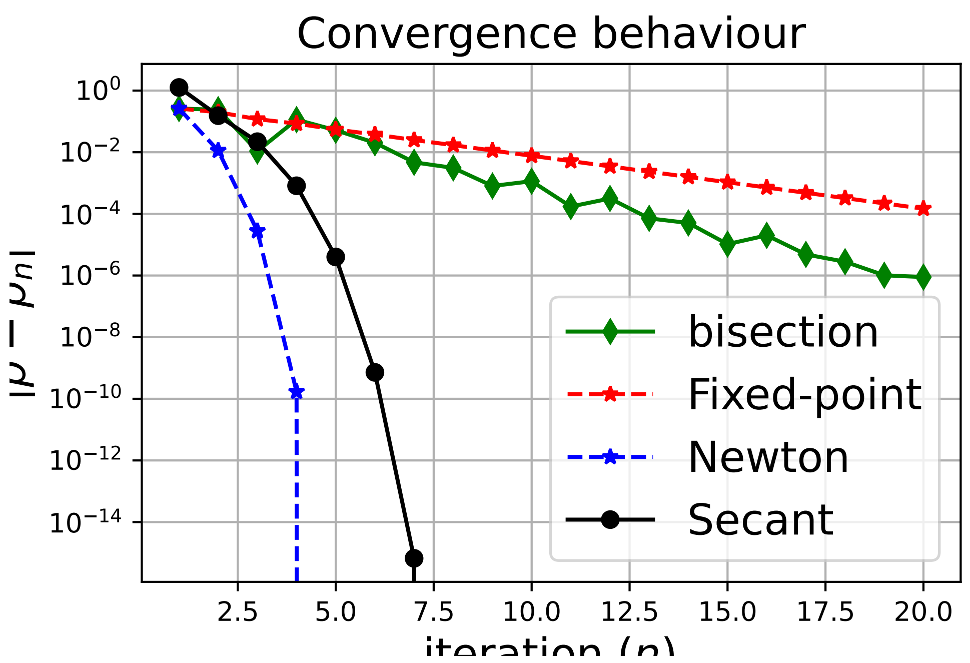


Figure 1.20: Convergence behaviour of Newton's method, Secant method and fixed point iteration

1.6 Order of convergence

1.6.1 Definitions

Definition 1.3 (Order of convergence)**.** Suppose $\{p_n\}_{n=1}^{\infty}$ is a sequence that converges to p with $p_n \neq p$ for all n. If positive constants λ and $\alpha \geq 1$ exist with

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

then $\{p_n\}_{n=1}^{\infty}$ is said to converge to p of order α, with asymptotic error constant λ.

Recall the definition of the absolute error: $E_n = |p_n - p|$ Intuitively when $\{p_n\}_{n=1}^{\infty}$ has order of convergence α, it means that

$$E_{n+1} \approx \lambda E_n^{\alpha} \tag{1.17}$$

for sufficiently large n.

If p_n converges to p as $n \to \infty$, then $E_n \to 0$. Thus, from Equation 1.17 we notice:

- The larger the α and the smaller the λ the faster the sequence converges.
- Note that a converging sequence cannot have order of convergence for $\alpha < 1$. If it did, we would have $E_{n+1} \approx \lambda E_n^{\alpha} \to \infty$ which is a contradiction to the fact that $E_n \to 0$.

Remark 1.2 (Linear and quadratic convergence)*.*

- When $\alpha = 1$ and $\lambda < 1$, $\{p_n\}_{n=1}^{\infty}$ is said to have **linear** order of convergence.

Since $E_{n+1} \approx \lambda E_n$, then $\lambda < 1$ otherwise the error will increase which contradicts the fact that $E_n \to 0$

- When $\alpha = 2$ then $\{p_n\}_{n=1}^{\infty}$ is said to have **quadratic** order of convergence.

If $|p| \neq 0$ we can divide Equation 1.17 to obtain an expression for the relative error:

$$\frac{E_{n+1}}{|p|} \approx \lambda \frac{E_n^{\alpha}}{|p|} = \lambda \frac{E_n^{\alpha}}{|p|^{\alpha}} |p|^{\alpha-1} \Longrightarrow RE_{n+1} = \lambda\, RE_n^{\alpha}\, |p|^{\alpha-1} \tag{1.18}$$

and thus

$$-\log_{10} RE_{n+1} = -\alpha \log_{10}(\,RE_n) - \log_{10}(\lambda\, |p|^{\alpha-1}) \tag{1.19}$$

Let us now recall that $-\log_{10} RE_n$ provides an approximation of the number of significant digits of the approximation at the nth iteration. Therefore, from Equation 1.19 we have the following:

ℹ Note

- For linear convergence ($\alpha = 1$), $-\log_{10} RE_{n+1} = -\log_{10}(\,RE_n) - \log_{10}(\lambda)$ and so each iteration has the a fixed number more of significant digits of accuracy than the previous.
- For quadratic convergence ($\alpha = 2$), $-\log_{10} RE_{n+1} = -2\log_{10}(\,RE_n) - \log_{10}(\lambda\,|p|)$, thus each iteration has (approximately) double the number of significant digits of the previous one.

Remark 1.3 (Superlinear Convergence)*.* A sequence $\{p_n\}_{n=1}^{\infty}$ is said to be superlinearly convergent to p if

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

Note that if $p_n \to p$ of order $\alpha > 1$ then p_n converges superlinearly since:

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|} = \lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|^{\alpha}} \frac{|p_{n+1} - p|^{\alpha}}{|p_n - p|} = \overbrace{\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|^{\alpha}}}^{\lambda} \underbrace{\lim_{n\to\infty} |p_{n+1} - p|^{\alpha-1}}_{0}$$

Exercise 1.15.

1. Consider the sequence defined by $p_{n+1} = \frac{1}{2}p_n$. Show that p_n converges to $p = 0$ linearly. Obtain an expression for $\log_{10}(E_n)$.

2. Consider the sequence defined by $p_{n+1} = \frac{1}{2}p_n^2$. Show that p_n converges to $p = 0$ quadratically.Obtain an expression for $\log_{10}(E_n)$.

3. For both sequences above, compute the values of the first 6 iterations and plot the absolute error versus n in a semilog plot.

Solution

1. Note that
$$p_n = \frac{1}{2}p_{n-1} = \frac{1}{2^2}p_{n-2} = \frac{1}{2^3}p_{n-3} = \cdots = \left(\frac{1}{2}\right)^n p_0$$

Since $\lim_{n\to\infty}\left(\frac{1}{2}\right)^n \to 0$ we have that $p_n \to 0$.
Let us look at the error:
$$|p_{n+1} - 0| = \frac{1}{2}|p_n - 0| \Longrightarrow \frac{|p_{n+1} - 0|}{|p_n - 0|} = \frac{1}{2}$$
so the convergence is linear with $\lambda = 1/2$.
$$\log_{10}|p_n - 0| = \log_{10}\left[\left(\frac{1}{2}\right)^n |p_0|\right] = \log_{10}|p_0| - n\log_{10}(2)$$

2. Note that
$$p_n = \frac{1}{2}p_{n-1}^2 = \frac{1}{2}\left(\frac{1}{2}\right)^2 p_{n-2}^4 = \frac{1}{2}\left(\frac{1}{2}\right)^2\left(\frac{1}{2}\right)^4 p_{n-3}^8 = \left(\frac{1}{2}\right)^{2^0+2^1+2^2} p_{n-3}^8 = \left(\frac{1}{2}\right)^{\sum_{i=0}^{n-1} 2^i} p_0^{2^n}$$

Note that
$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

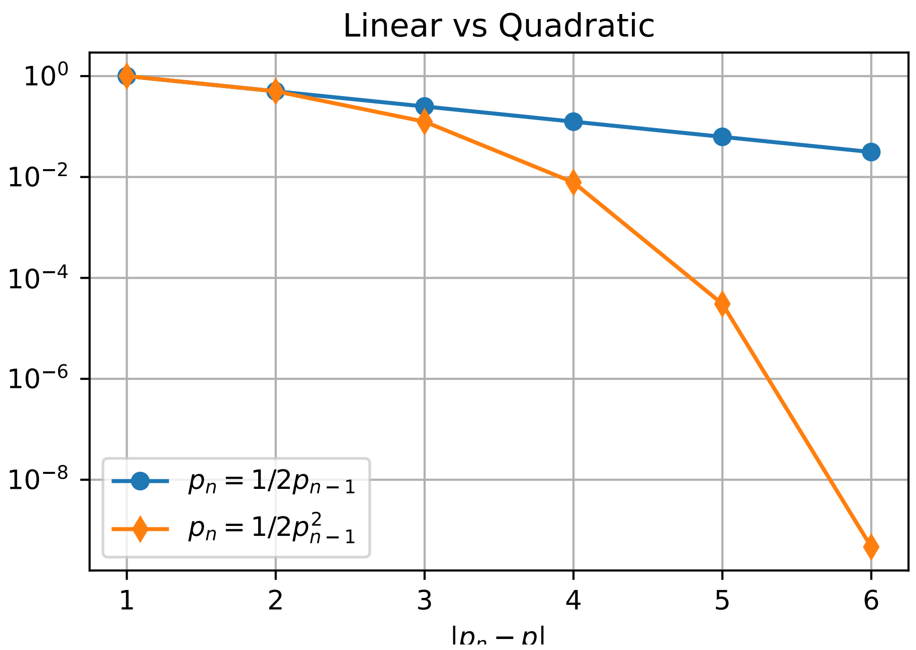
so
$$p_n = \left(\frac{1}{2}\right)^{2^n-1} p_0^{2^n} = 2\left(\frac{1}{2}\right)^{2^n} p_0^{2^n} = 2\left(\frac{p_0}{2}\right)^{2^n}$$

If $n \to \infty$ then $2^n \to \infty$. Therefore, $p_n \to 0$ provided that $|p_0| < 2$.
Furthermore,
$$\frac{|p_{n+1} - 0|}{|p_n - 0|^2} = \frac{1}{2},$$
thus the convergence is quadratic with $\lambda = 1/2$. We also note that
$$\log_{10}|p_n - 0| = \log_{10}\left[2\left(\frac{p_0}{2}\right)^{2^n}\right] = 2^n\log_{10}\left[\frac{|p_0|}{2}\right] + \log_{10}(2).$$

```
import numpy as np
import matplotlib.pyplot as plt

p0=1.0
Nmax=6
p_linear=np.zeros(Nmax,)
p_quadratic=np.zeros(Nmax,)
p_linear[0]=p0
p_quadratic[0]=p0

for i in range(1,Nmax):
    p_linear[i]=0.5* p_linear[i-1]
    p_quadratic[i]=0.5* p_quadratic[i-1]**2
    print(f"Error (linear) = {np.abs(p_linear[i]):.16f}\t Error (quadratic)
     ↪  = {np.abs(p_quadratic[i]):.16f}")
```

```
Error (linear) = 0.5000000000000000  Error (quadratic) = 0.5000000000000000
Error (linear) = 0.2500000000000000  Error (quadratic) = 0.1250000000000000
Error (linear) = 0.1250000000000000  Error (quadratic) = 0.0078125000000000
Error (linear) = 0.0625000000000000  Error (quadratic) = 0.0000305175781250
Error (linear) = 0.0312500000000000  Error (quadratic) = 0.0000000004656613
```

```
n= np.arange(1,Nmax+1)

fig, ax = plt.subplots()
ax.set_yscale("log")
ax.set_xlabel("n")
ax.set_xlabel("$| p_{n}-p | $")
ax.set_title("Linear vs Quadratic")
ax.grid(True)

# Plot error
ax.plot(n, np.abs(p_linear) , "o",
 ↪  label=r"$p_{n}=1/2p_{n-1}$",linestyle="-")
ax.plot(n, np.abs(p_quadratic) , "d",
 ↪  label=r"$p_{n}=1/2p_{n-1}^2$",linestyle="-")
ax.legend()
```



1.6.2 Order of convergence for fixed point iteration

The aim of this subsection is to prove that fixed-point iteration converges to a fixed point with order of convergence $\alpha = 1$ (i.e. linearly) according to definition Definition 1.3. Note that the previous observations that we made of linear convergence of the fixed-point iteration method were done based on formal arguments while here we make a rigorous proof.

Theorem 1.6. *Let $g \in C[a,b]$ be such that $g(x) \in [a,b]$ for all $x \in [a,b]$. Suppose, in addition that $g'(x)$ is* ***continuous*** *on (a,b) and that a constant $0 < k < 1$ exists with*

$$|g'(x)| \leq k, \qquad \textit{for all} \;\; x \in (a,b),$$

If $g'(p) \neq 0$ then, for any number $p_0 \neq p$ in $[a,b]$, the sequence defined by

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

converges only linearly to the unique fixed point $p \in [a,b]$.

Proof. Note that

$$p_{n+1} - p = \overbrace{g(p_n) - g(p) = g'(\xi_n)(p_n - p)}^{Mean-value\ Theorem}$$

where $\xi_n \in [p_n, p]$. Note that $\xi_n \to p$ as $n \to \infty$ since $p_n \to p$. Then:

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|} = \lim_{n\to\infty} \frac{|g'(\xi_n)(p_n - p)|}{|p_n - p|} = \lim_{n\to\infty} |g'(\xi_n)| = |g'(p)| \tag{1.20}$$

Since $g'(p) \neq 0$, the sequence p_n converges linearly (i.e. $\alpha = 1$) with $\lambda = |g'(p)|$. □

Note

From Equation 1.20 we note that p_n cannot converge superlinearly to p unless $\lambda = |g'(p)| = 0$

As we see in the following theorem, when $g'(p) = 0$ and $g \in C^2[a, b]$ fixed-point iteration converges with quadratic order of convergence.

Theorem 1.7 (Quadratic convergence of fixed-point iteration). *Let p be a solution of the equation $x = g(x)$. Suppose that $g'(p) = 0$ and g'' is continuous with*

$$|g''(x)| < M \tag{1.21}$$

on an open interval I containing p. Then there exists a $\delta > 0$ such that for $p_0 \in [p - \delta, p + \delta]$, the sequence defined by $p_n = g(p_{n-1})$, when $n \geq 1$, converges at least quadratically to p. Moreover, for sufficiently larger values of n:

$$|p_{n+1} - p| < \frac{M}{2}|p_n - p|^2 \tag{1.22}$$

Proof. We would first like to apply Theorem 1.4 to g on the interval $[p - \delta, p + \delta]$ to conclude that the fixed point iteration converges.

Since $g'(p) = 0$, from the continuity of g' we can choose δ such that

$$|g'(x)| \leq k < 1 \qquad \text{for all} \quad x \in [p - \delta, p + \delta] \subset I$$

.

We now need to show that $g(x) \in [p - \delta, p + \delta]$ for all $x \in [p - \delta, p + \delta]$.

In other words, we need to show that $|g(x) - p| \leq \delta$ for all x such that $|p - x| \leq \delta$.

Note that

$$|g(x) - p| = |g(x) - g(p)| = |g'(\xi)(p - x)| = |g'(\xi)||p - x| \tag{1.23}$$

for some ξ between p and x. Since $x \in [p - \delta, p + \delta]$, then $\xi \in [p - \delta, p + \delta]$ and so $|g'(\xi)| < k < 1$.

Hence Equation 1.23 becomes

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

Thus $|p - x| \leq \delta \implies |g(x) - p| \leq \delta$

From Theorem 1.4 we conclude that the sequence defined by $p_{n+1} = g(p_n)$ converges to p (the unique fixed point) provided that $p_0 \in [p - \delta, p + \delta]$.

From Taylor's Theorem Theorem A.3 we have:

$$g(p_n) = g(p) + g'(p)(p_n - p) + \frac{g''(\xi_n)}{2}(p_n - p)^2$$

for some ξ_n between p and p_n. From our assumptions we know that $p = g(p)$ (i.e. p is a fixed point) and $g'(p) = 0$. Then,

$$p_{n+1} = g(p_n) = p + \frac{g''(\xi)}{2}(p_{n-1} - p)^2 \Longrightarrow |p_{n+1} - p| = \left|\frac{g''(\xi_n)}{2}(p_{n-1} - p)^2\right|$$

Thus

$$|p_{n+1} - p| = \left|\frac{g''(\xi_n)}{2}\right| |p_{n-1} - p|^2$$

Since ξ_n is between p and p_n and $p_n \to p$, then $\xi_n \to p$ as $n \to \infty$. Therefore,

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_{n-1} - p|^2} = \lim_{n\to\infty} \left|\frac{g''(\xi_n)}{2}\right| = \left|\frac{g''(p)}{2}\right| < \frac{M}{2}$$

where we have used Equation 1.21. □

Consider the function

$$g(x) = \frac{1}{2}\left(x + \frac{1}{x}\right) \tag{1.24}$$

on the interval $[1/2, 3]$ and note that $p = 1$ is a fixed point of g (see Figure 1.21).

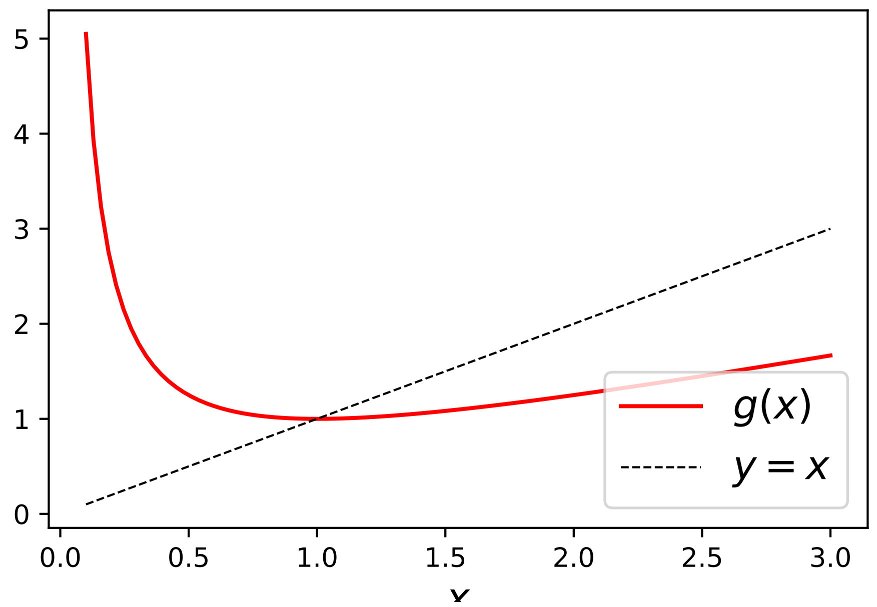


Figure 1.21: Example of quadratic convergence of fixed-point iteration

Note that

$$g'(x) = \frac{1}{2}\left(1 - \frac{1}{x^2}\right)$$

and so $g'(p) = 0$ for the fixed point $p = 1$. Furthermore,

$$g''(x) = \frac{1}{x^3}$$

and thus since $1/2 \le x$, then $0 < 1/x \le 2$ for all $x \in [1/2, 3]$ and so

$$|g''(x)| = \left|\frac{1}{x^3}\right| = \frac{1}{x^3} \le 8$$

Therefore, from Theorem 1.7 we expect that fixed-point iteration will convergence quadratically if we start sufficiently close to $p = 1$. Indeed, if we start for example with $p_0 = 3$, the error that we obtain using fixed-point iteration is shown in Figure 1.22.

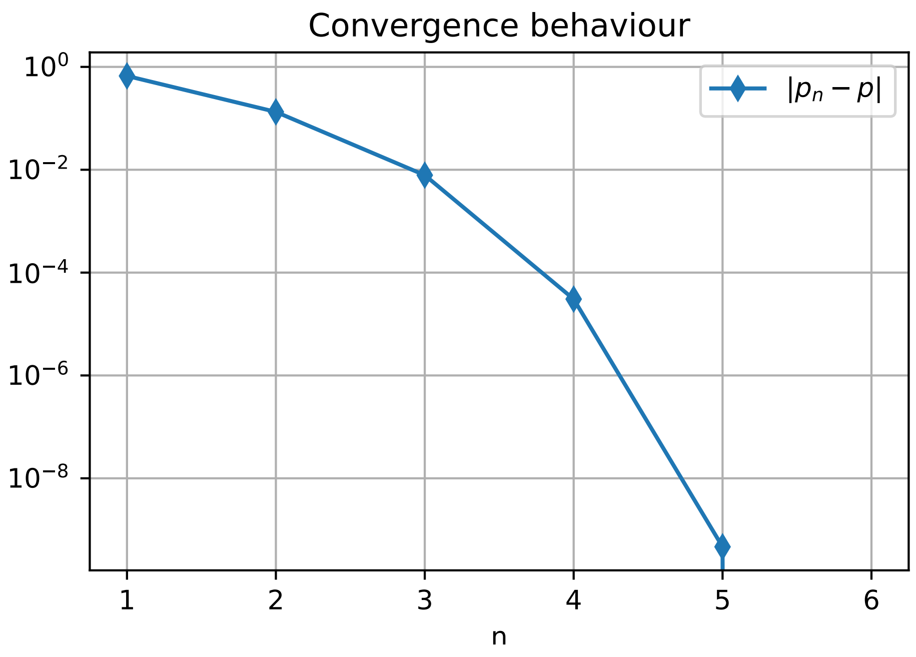


Figure 1.22: Quadratic convergence

! Important 11: Computing Activity

Use the codes you developed for Section Section 1.3 to reproduce Figure 1.22 for the function Equation 1.24.

Having established the conditions for which fixed point iteration displays quadratic convergence, we can now show that under suitable conditions Newton's method converges quadratically

Corollary 1.2. *Newton's method applied to a sufficiently smooth f converges quadratically to a root of f if $f'(p) \neq 0$.*

Proof. We apply Theorem 1.7 to $g(x) = x - \frac{f(x)}{f'(x)}$ which, as noted earlier enable us to write Newton's method as a fixed point iteration. Note that $g'(p) = 0$ since $f'(p) \neq 0$ (see Equation 1.15) and $g''(p)$ exists if f is sufficiently smooth. □

1.6.3 Additional problems

Exercise 1.16. Let $f(x) = x^2 - 6$.

1. Use Newton's method with $p_0 = 1$ to find p_2.
2. Use the Secant method with $p_0 = 3$ and $p_1 = 2$ to find p_3.

Solution

1. $p_2 \approx 2.60714$.
2. $p_3 \approx 2.45454$.

Exercise 1.17. Assume that the hypothesis of Theorem 1.5 hold true. Suppose p_{n-1} is an approximation to the root p of f. From Taylor's Theorem (Theorem A.3) we have:

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

where ξ is a point between p_{n-1} and p.

Show that if $f'(p) \neq 0$, Newton's method can be derived when one assumes that $(p - p_{n-1})^2$ is small enough to be neglected.

Solution

Using the fact that p is the root of f (i.e. $f(p) = 0$) we have

$$0 = f(p_{n-1}) + f'(p_{n-1})(p - p_{n-1}) + \frac{f''(\xi)}{2}(p - p_{n-1})^2$$

which we can solve for p:

$$p = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})} - \frac{f''(\xi)}{2f'(p_{n-1})}(p - p_{n-1})^2 \tag{1.25}$$

where we have also assumed that $f'(p_{n-1}) \neq 0$ (recall from the prood of Theorem **??** that we can choose some interval for which $f(p_n) \neq 0$ for all $n = 0, 1, ...$).
Now, If $|p - p_{n-1}|$ is small then $(p - p_{n-1})^2$ is even smaller and so we can neglect the quadratic term in the previous expression:

$$p \approx p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}$$

and so we obtain the approximation p_n defined via Equation 1.12, i.e. Newton's method.

Exercise 1.18. Show that the following sequences converge linearly to $p = 0$. How large must n be before $|p_n - p| \leq 5 \times 10^{-2}$.

1. $p_n = \frac{1}{n}$, $n \geq 1$
2. $p_n = \frac{1}{n^2}$, $n \geq 1$

Solution

Clearly both sequences converge to $p = 0$. To prove linear convergence we need to show that

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

.

1. Note that

$$\frac{|p_{n+1} - p|}{|p_n - p|} = \frac{|1/(n+1)|}{|1/n|} = \frac{1}{1 + \frac{1}{n}} \tag{1.26}$$

Note that

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

However, from Equation 1.26 we have that

$$\frac{|p_{n+1} - p|}{|p_n - p|} = \frac{1}{1 + \frac{1}{n}} < 1 \quad \forall n \geq 1$$

So even though $\lambda = 1$ above, the error is always garanteed to reduce.
To reach 5×10^{-2}, we need

$$\frac{1}{n} \leq 0.05 \quad \Rightarrow \quad n \geq 20$$

2.
Similarly:

$$\frac{|p_{n+1} - p|}{|p_n - p|} = \frac{|1/(n+1)^2|}{|1/n^2|} = \frac{1}{1 + \frac{2}{n} + \frac{1}{n^2}} \to 1 \qquad \text{as } n \to \infty$$

To reach 5×10^{-2}, we need

$$\frac{1}{n^2} \leq 0.05 \quad \Rightarrow \quad n \geq \sqrt{20} \approx 4.47$$

hence n must be at least 5.

Exercise 1.19. Assuming the hypothesis of Theorem 1.5 hold, derive the following error formula for Newton's method

$$|p-p_n| \le \frac{M}{2|f'(p_{n-1})|}|p-p_{n-1}|^2$$

where $M = \max_{x\in[a,b]}|f''(x)|$

Solution

From Equation 1.25 we have

$$p = \overbrace{p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}}^{p_n} - \frac{f''(\xi)}{2f'(p_{n-1})}(p-p_{n-1})^2$$

and so

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

Thus

$$|p-p_n| = \left|\frac{f''(\xi)}{2f'(p_{n-1})}(p-p_{n-1})^2\right| = \frac{|f''(\xi)|}{2|f'(p_{n-1})|}|p-p_{n-1}|^2 \le \frac{M}{2|f'(p_{n-1})|}|p-p_{n-1}|^2$$

Exercise 1.20. It can be shown that if $\{p_n\}_{n=1}^{\infty}$ are convergent Secant approximations to a root p of $f(x)$, then a constant C exists with $|p_{n+1}-p| \approx C|p_n-p||p_{n-1}-p|$ for sufficiently large n. Assume $\{p_n\}_{n=1}^{\infty}$ converges to p of order α and show that $\alpha = (1+\sqrt{5})/2 \approx 1.62$.

Solution

Assume the convergence is of order α. Since $|p_{n+1}-p| \approx C|p_n-p||p_{n-1}-p|$ for large n, a key observation is that

$$\frac{|p_{n+1}-p|}{|p_n-p|^\alpha} \to \lambda \qquad \Rightarrow \qquad \frac{C|p_n-p||p_{n-1}-p|}{|p_n-p|^\alpha} \to \lambda\,.$$

Rearranging the final expression,

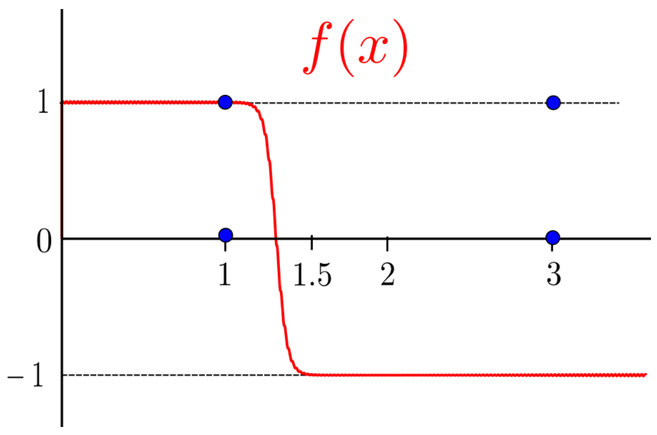
$$|p_n-p|^{1-\alpha}\,|p_{n-1}-p| \to \frac{\lambda}{C} \qquad \Leftrightarrow \qquad \frac{|p_n-p|}{|p_{n-1}-p|^{\frac{1}{\alpha-1}}} \to \left(\frac{\lambda}{C}\right)^{\frac{1}{1-\alpha}}\,.$$

But, as the convergence is of order α, it must also be true that

$$\frac{|p_n-p|}{|p_{n-1}-p|^\alpha} \to \lambda\,.$$

Therefore, $\alpha = \frac{1}{\alpha-1}$, in other words, $\alpha^2-\alpha-1=0$, which has positive root $\alpha = \frac{1}{2}+\frac{1}{2}\sqrt{5}$. (This is the so-called "golden ratio"). \ (Note also that $\lambda = (\frac{\lambda}{C})^{\frac{1}{1-\alpha}}$, which allows for the determination of λ, but this was not asked).

Exercise 1.21. Consider Bisection, Newton and Secant methods for finding the root of $f:[0,3.5]\to\mathbb{R}$ defined by the graph below:



Note that $f(x) = 1$ for $x \in [0, 1.1]$ and $f(x) = -1$ for $x > 1.3$.

Consider the two following starting values $p_0 = 1$ and $p_0 = 3$ for Bisection and Newton. Consider $p_0 = 1$, $p_1 = 3$ and $p_0 = 3$, $p_1 = 1$ for the secant method.

Which method(s) will converge?

Solution

- Bisection converges for any starting value in $[0, 3.5]$ since f is clearly continuous and $f(0) = 1$ and $f(3.5) = -1$ so $f(0)f(3.5) < 0$.
- Newton method fails for both $p_0 = 1$ and $p_0 = 3$ since $f'(p_0) = 0$.
- Secant method also fails since $f(p_{n-1}) = f(p_{n-2})$ for some n. Indeed, note that for $p_0 = 1$, $p_1 = 3$,

$$p_2 = p_1 - f(p_1)\frac{p_1 - p_0}{f(p_1) - f(p_0)} = 3 - (-1)\frac{3-1}{-1-1} = 3 - 1 = 2$$

and so $f(p_1) = f(p_2) = -1$
For the choice $p_0 = 3$, $p_1 = 1$ we also have $p_2 = 2$ (please check). In this case $f(p_1) = 1$ and $f(p_2) = -1$ so we can compute

$$p_3 = p_2 - f(p_2)\frac{p_2 - p_1}{f(p_2) - f(p_1)} = \frac{3}{2}$$

Since $f(p_2) = f(p_3) = -1$, p_4 is not well-defined.

Listing 1.2 `bisect_simple.py` Implementation of the bisection method as a function Algorithm 1.1

```
#import numpy library:

import numpy as np

def bisection_simple(f,a,b,Nmax):
    """
    Bisection Method: Returns a numpy array of the
    sequence of approximations obtained by the bisection method.

    Inputs:
    ----------
    f : function
        Input function for which the zero is to be found.
    a : float
        Left side of interval.
    b : float
        Right side of interval.
    Nmax : integer
        Number of iterations to be performed.

    Returns
    -------
    p_array : numpy.ndarray,
            Array containing the sequence of approximations.
            The shape is (Nmax,)
    """

    #Initialise numpy array of size Nmax
    p_array=np.zeros(Nmax,)

    #Begin bisection method:
    fa=f(a)
    for n in range(Nmax):
        p=(a+b)/2   # midpoint
        fp=f(p)     # evaluate f at midpoint
        #define new interval
        if fp*fa>0:
            a=p
            fa=fp
        else:
            b=p
        p_array[n]=p
    return p_array

if __name__ == "__main__":
    #function
    f = lambda x: x**3+x**2-2*x-2
    #interval
    a, b = 1.0, 2.0
    #maximum number of iterations
    Nmax = 10

    #run Bisection Method:
    bisec_approx=bisection_simple(f,a,b,Nmax)
    print("Approximations are:\n", bisec_approx)
```

```
Approximations are:
 [1.5        1.25       1.375      1.4375     1.40625    1.421875
 1.4140625  1.41796875 1.41601562 1.41503906]
```

Listing 1.3 `rootsolvers.py` Bisection method as a function with option to test convergence

```python
#import numpy library:
import numpy as np

def bisection(f,a,b,Nmax,pe=None):
    """
    Bisection Method: Returns a numpy array of the
    sequence of approximations obtained by the bisection method.

    Inputs:
    ----------
    f : function
        Input function for which the zero is to be found.
    a : float
        Left side of interval.
    b : floaat
        Right side of interval.
    Nmax : integer
        Number of iterations to be performed.
    pe : float
        `exact` solution if available (default = None)

    Returns
    -------
    p_array : numpy.ndarray,
            Array containing the sequence of approximations.
            The shape is (Nmax,),
    """

    #check if we know there is a root
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) and f(b) must have different signs!")

    # Initialise the array with zeros
    p_array = np.zeros(Nmax)



    fa=f(a)
    for n in range(Nmax):
        p=(a+b)/2  # midpoint
        fp=f(p)    # evaluate f at midpoint
        #if we have exact solution compute errors
        if pe is not None:
            E=np.abs(p-pe)           #Compute error
            RE=np.abs(p-pe)/np.abs(pe) # relative error
            print(f"n {n+1} \t p_n:{p:.8f} \t E_n:{E:.8f} \t RE_n:{RE:.8f} \t -log10(RE_n):{-np.log10(RE):.2f}")
             ↪

        #define new interval
        if fp*fa>0:
            a=p
            fa=fp
        else:
            b=p

        p_array[n]=p


    return p_array

if __name__ == "__main__":

  #function
  f = lambda x: x**3+x**2-2*x-2
  #interval
  a, b = 1.0, 2.0
  #maximum number of iterations
  Nmax = 10
  #exact solution:
  pe=np.sqrt(2)
  #run Bisection Method:
  bisec_approx=bisection(f,a,b,Nmax,pe)
  print("Approximations are:\n", bisec_approx)
  print(f"the root of f to 16 significant figures is {pe:.16}")
```

```
n 1      p_n:1.50000000      E_n:0.08578644      RE_n:0.06066017     -log10(RE_n):1.22
n 2      p_n:1.25000000      E_n:0.16421356      RE_n:0.11611652     -log10(RE_n):0.94
n 3      p_n:1.37500000      E_n:0.03921356      RE_n:0.02772818     -log10(RE_n):1.56
n 4      p_n:1.43750000      E_n:0.02328644      RE_n:0.01646600     -log10(RE_n):1.78
n 5      p_n:1.40625000      E_n:0.00796356      RE_n:0.00563109     -log10(RE_n):2.25
n 6      p_n:1.42187500      E_n:0.00766144      RE_n:0.00541745     -log10(RE_n):2.27
n 7      p_n:1.41406250      E_n:0.00015106      RE_n:0.00010682     -log10(RE_n):3.97
n 8      p_n:1.41796875      E_n:0.00375519      RE_n:0.00265532     -log10(RE_n):2.58
n 9      p_n:1.41601562      E_n:0.00180206      RE_n:0.00127425     -log10(RE_n):2.89
n 10     p_n:1.41503906      E_n:0.00082550      RE_n:0.00058372     -log10(RE_n):3.23
Approximations are:
 [1.5        1.25       1.375      1.4375     1.40625    1.421875
 1.4140625  1.41796875 1.41601562 1.41503906]
the root of f to 16 significant figures is 1.414213562373095
```

Algorithm 1.3 Fixed Point Iteration

```
1: **Inputs:** $g \in C[a,b]$ such that $g(x) \in [a,b]$ for all $x \in [a,b]$,
2:    $p_0$: Initial approximation
3:    $N_{\max}$: Maximum number of iterations
4: **Output:** $p$: approximation to the fixed point (solution of $g(p) = p$).
5: procedure FixedPointIteration($g, p_0, N_{max}$)
6:     Set $p = p_0$
7:     for $n = 1$ to $N_{max}$ do
8:         $p = g(p)$
9:     end for
10: end procedure
```

Listing 1.6 Implementation of fixed-point iteration

```python
import numpy as np

def FixedPointIteration(g,p0,Nmax):
    """
    Fixed-point iteration Method: Returns the approximations obtained by fixed
 ↪  point
    iteration after Nmax iteration.

    Inputs:
    ----------
    g : function
        Input function for which the fixed point is to be found.
    p0 : floating point
        Initial approximation
    Nmax : integer
        Number of iterations to be performed.

    Returns
    -------
    p : approximation after Nmax iterations
    """
    p=p0
    for n in range(Nmax):
        p=g(p)
        print(f"iteration {n+1} \t p_n:{p:5.16f}\t p_n-g(p_n):{p-g(p):5.16f}")
    return p
```

Algorithm 1.4 Fixed Point Iteration with stopping criteria

```
1: Inputs: $g \in C[a,b]$ such that $g(x) \in [a,b]$ for all $x \in [a,b]$,
2:    $p_0$: initial approximation.
3:    $N_{\max}$: Max no. of iterations
4:    TOL: desired level of accuracy
5: Output: $p$: approximation to the solution of $g(p) = p$.
6: procedure FixedPointWithStoppingCriteria($g, p_0, N_{max}$, TOL)
7:     Set $n = 1$
8:     while $n \le N_{max}$ do
9:         Compute $p = g(p_0)$
10:        if $|p - p_0| <$ TOL then
11:            break
12:            Output $(p)$
13:        end if
14:        set $n = n + 1$
15:        set $p_0 = p$
16:    end while
17: end procedure
```

Algorithm 1.5 Newton's Method

```
1: **Inputs:** $f \in C^2[a, b]$,
2:     $p_0$: Initial approximation
3:     $N_{max}$: Number of iterations
4: **Output:** $p$: approximation to the solution of $f(p) = 0$ (after $N_{max}$ iterations).
5: **procedure** NewtonMethod($f, p_0, N_{max}$)
6:     Set $n = 1$
7:     **while** $n \le N_{max}$ **do**
8:         $p = p_0 - f(p_0)/f'(p_0)$
9:         set $n = n + 1$
10:        set $p_0 = p$
11:    **end while**
12: **end procedure**
```

Listing 1.10

```python
#import numpy library:

import numpy as np

def Newton_simple(f,df, p0,Nmax):
    """
    Newton's Method: Returns a numpy array of the
    sequence of approximations obtained by  Newton's method.

    Inputs:
    ----------
    f : function
        Input function for which the zero is to be found.
    df : function
        Derivative of the input function.
    Nmax : integer
        Number of iterations to be performed.

    Returns
    -------
    p_array : numpy.ndarray,
            Array containing the sequence of approximations.
            The shape is (Nmax,)
    """
    n=0
    #initialise array to zero
    p=np.zeros(Nmax,)
    while n<Nmax:
        p0=p0-f(p0)/df(p0) #update
        p[n]=p0 #save update to array
        n=n+1 #need to update n
    return p
```

Algorithm 1.6 Secant Method

```
1: Inputs: $f \in C^2[a,b]$,
2:     $p_0$ and $p_1$: Initial approximations,
3: $N_{\max}$: Max number of iterations
4: Output: $p$: approximation to the root of $f$.
5: procedure SecantMethod($f, p_0, N_{max}$)
6:     Set: $n = 2$, $q_0 = f(p_0)$, $q_1 = f(p_1)$
7:     while $n \leq N_{max}$ do
8:         Set $p = p_1 - q_1 \frac{(p_1 - p_0)}{(q_1 - q_0)}$
9:         set $n = n + 1$
10:        set $p_0 = p_1$, $q_0 = q_1$, $p_1 = p$, $q_1 = f(p)$
11:    end while
12: end procedure
```

Chapter 2

Direct Methods

In this section we study numerical methods for solving a linear system of n equations in n variables of the form

$$\begin{aligned}
E_1 : &\quad a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
E_2 : &\quad a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\
\vdots &\quad \vdots \\
E_n : &\quad a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n
\end{aligned} \tag{2.1}$$

Given the constants $\{a_{ij}\}_{i,j=1}^{n}$ and $\{b_i\}_{i=1}^{n}$ the goal is to compute the independent variables $x_1, \dots, x_n$.

The system above can be written in matrix-vector form as $A\mathbf{x} = \mathbf{b}$ where

$$\overbrace{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & \dots & a_{nn} \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}^{\mathbf{x}} = \overbrace{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}}^{\mathbf{b}} \tag{2.2}$$

Direct Methods are aimed at computing the exact solution of system given in (Equation 2.2) in a **finite** number of steps.

Note 3: Note

Recall from previous modules that

1. A square matrix is (i.e. $n \times n$) is called non-singular if its inverse (denoted by A^{-1}) exists. Note that $AA^{-1} = A^{-1}A = I$ where I is the identity matrix:

$$I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & 0\dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

2. A square matrix A is non-singular if an only if its determinant (denoted by $\det(A)$) is not zero
3. A is non-singular if and only if the system $A\mathbf{x} = \mathbf{b}$ has a unique solution for any $\mathbf{b}$.

2.1 Gaussian Elimination

2.1.1 Motivation

Let us consider the problem of solving the system of linear equations:

$$\begin{aligned}
E_1 : &\quad 2x_1 + 3x_2 + 4x_3 &= 2 \\
E_2 : &\quad 4x_1 + 9x_2 + 10x_3 &= 6 \\
E_3 : &\quad 6x_1 + 15x_2 + 20x_3 &= 12
\end{aligned} \tag{2.3}$$

which we can write in the matrix-vector form $A\mathbf{x} = \mathbf{b}$ as follows:

$$\overbrace{\begin{bmatrix} 2 & 3 & 4 \\ 4 & 9 & 10 \\ 6 & 15 & 20 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}^{\mathbf{x}} = \overbrace{\begin{bmatrix} 2 \\ 6 \\ 12 \end{bmatrix}}^{\mathbf{b}} \tag{2.4}$$

Recall from your previous modules that, in order to solve linear systems (i.e. to find $\mathbf{x}$) like the one above, we can conduct row operations on A and $\mathbf{b}$ to transform the system into a (reduced) form that can be easily solved. To be more precise, let us first augment the system

$$\tilde{A} = [A \,|\, \mathbf{b}] = \left[\begin{array}{ccc|c} 2 & 3 & 4 & 2 \\ 4 & 9 & 10 & 6 \\ 6 & 15 & 20 & 12 \end{array}\right]$$

and in what follows we denoted the rows by E_1,E_2 and E_3 while the entries of $\tilde{A}$ are denoted by a_{ij}.

Our aim is to perform operations on E_1,E_2 and E_3 to obtain the reduced form of the system which has zeros below the diagonal. While we are allowed to performed any row operations, here we apply the same approach on each step. In the first step we leave E_1 unchanged and aim is to make all terms a_{21} and a_{31} in the column below the leading term of E_1 (i.e. $a_{11} = 2$) equal to zero. To this end we conduct

$$E_2 \rightarrow E_2 - (a_{21}/a_{11})E_1 = \overbrace{[4 \quad 9 \quad 10 \quad 6]}^{E_2} - (4/2)\overbrace{[2 \quad 3 \quad 4 \quad 2]}^{E_1} = [0 \quad 3 \quad 2 \quad 2]$$
$$E_3 \rightarrow E_3 - (a_{31}/a_{11})E_1 = \overbrace{[6 \quad 15 \quad 20 \quad 12]}^{E_3} - (6/2)\overbrace{[2 \quad 3 \quad 4 \quad 2]}^{1} = [0 \quad 6 \quad 8 \quad 6]$$

The matrix that we obtain is $\tilde{A}^{(2)}$ in Figure 2.1. Note that the entries of the rows E_2 and E_3 are different from those in $\tilde{A}^{(2)}$ bu for simplicity we use the same notation.

For the next step we leave E_1 and E_2 unchanged and try to make the entry a_{32} below the diagonal term in E_2 (i.e. a_{22}). We can do this applying the same procedure, i.e.

$$E_3 \rightarrow E_3 - (a_{32}/a_{22})E_2 = \overbrace{[0 \quad 6 \quad 8 \quad 6]}^{E_3} - (6/3)\overbrace{[0 \quad 3 \quad 2 \quad 2]}^{E_2} = [0 \quad 0 \quad 4 \quad 2]$$

Hence the final matrix in reduced form is $\tilde{A}^{(3)}$ shown in Figure 2.1.

$$\begin{array}{r} E_1: \\ E_2: \\ E_3: \end{array} \overbrace{\left[\begin{array}{ccc|c} \textcircled{2} & 3 & 4 & 2 \\ 4 & 9 & 10 & 6 \\ 6 & 15 & 20 & 12 \end{array}\right]}^{A = \tilde{A}^{(1)}} \sim \begin{array}{r} E_1 \rightarrow E_1 \\ E_2 - (4/2)E_1 \rightarrow E_2 \\ E_3 - (6/2)E_1 \rightarrow E_3 \end{array} \overbrace{\left[\begin{array}{ccc|c} 2 & 3 & 4 & 2 \\ 0 & \textcircled{3} & 2 & 2 \\ 0 & 6 & 8 & 6 \end{array}\right]}^{\tilde{A}^{(2)}} \sim \begin{array}{r} E_1 \rightarrow E_1 \\ E_2 \rightarrow E_2 \\ E_3 - (6/3)E_2 \rightarrow E_3 \end{array} \overbrace{\left[\begin{array}{ccc|c} 2 & 3 & 4 & 2 \\ 0 & 3 & 2 & 2 \\ 0 & 0 & 4 & 2 \end{array}\right]}^{\tilde{A}^{(3)}}$$

Figure 2.1: Example of Gaussian elmination

Note the two steps that we applied above can be written as

$$E_j \rightarrow E_j - (a_{ji}/a_{ii})E_i, \qquad \text{for} \quad j = i+1, \dots, n$$

for $i = 1, 2$ and $n = 3$. At each step of the process, the diagonal term a_{ii}, below which we aim to create zeros, is referred to as **pivot element**.

Because row operations do not change the solution to the system given in Equation 2.3, once we arrive at the reduced form, we can find the solution by solving the equivalent system:

$$\begin{aligned} 2x_1 + 3x_2 + 4x_3 &= 2 \\ 3x_2 + 2x_3 &= 2 \\ 4x_3 &= 2. \end{aligned}$$

This can be done systematically via **backward- substitution**, i.e.

$$\begin{aligned} x_3 &= \frac{2}{4} = \frac{1}{2} \\ x_2 &= \frac{1}{3}\Big(2 - 2x_3\Big) = \frac{1}{3}(2-1) = \frac{1}{3} \\ x_1 &= \frac{1}{2}\Big(2 - 4x_3 - 3x_2\Big) = \frac{1}{2}\Big(2 - \frac{4}{2} - \frac{3}{3}\Big) = -\frac{1}{2} \end{aligned} \tag{2.5}$$

The two combined procedures that we have followed above is called **Gaussian (forward) elimination with backward substitution**. The remaining of this section is aimed at deriving this algorithm for the general system given in (Equation 2.2).

Numpy arrays

Please make sure to review Python Year 1 notes how to work with matrices with numpy. In the lines below we verify that Equation 2.5 is indeed the solution to the system Equation 2.4.

```
import numpy as np

A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)
b=np.array([[2],[6],[12]], dtype=float)
x=np.array([[-1/2],[1/3],[1/2]], dtype=float)

print(f"A times x =\n {A@x}")
print(f"A times x -b  =\n {A@x-b}")
```

```
A times x =
 [[ 2.]
 [ 6.]
 [12.]]
A times x -b  =
 [[0.]
 [0.]
 [0.]]
```

2.1.2 Gaussian elimination.

Consider the $n \times n$ system from Equation 2.2. We start with the augmented matrix $\tilde{A} = [A\,|\mathbf{b}]$, where for ease in the notation we relabel the entries of $\mathbf{b}$ by defining $a_{i,n+1} \equiv b_i$ for $i = 1, \dots n$. Therefore, the entries of $\tilde{A}$ are $[\tilde{A}]_{i,j} = a_{ij}$ for $1 \le i \le n$, $1 \le j \le n+1$.

Let us conduct the first step of Gaussian elimination to transform the original matrix $\tilde{A}^{(1)} \equiv \tilde{A}$ into a matrix $\tilde{A}^{(2)}$ with zeros below the first entry of E_1 (inside $\circ$ shown in Figure 2.2). This requires us to make the following operations

$$E_j - (a_{j1}^{(1)}/a_{11}^{(1)})E_1 \to E_j \qquad j = 2, 3, \dots, n$$

as shown in Figure 2.2. Notice that have included the superscript $^{(1)}$ in all elements of $\tilde{A}^{(1)}$. The entries from $E_2 \dots E_n$ in $\tilde{A}^{(2)}$ are different from those in $\tilde{A}^{(1)}$ and thus these entries have superscript $^{(2)}$.

We now repeat the process to transform $\tilde{A}^{(2)}$ to $\tilde{A}^{(3)}$ with zeros below the entry a_{22} inside $\circ$. We thus conduct

$$E_j - (a_{j2}^{(2)}/a_{22}^{(2)})E_2 \to E_j \qquad j = 3, 4, \dots n.$$

We can perform the procedure above sequentially noticing that the general term for the $\tilde{A}^{(i)}$ (for $2 \le i < n$) given in Figure 2.3 where we also show that the transition to $\tilde{A}^{(i+1)}$ requires us to conduct:

$$E_j - (a_{ji}^{(i)}/a_{ii}^{(i)})E_i \to E_j \qquad j = i+1, \dots n.$$

The final matrix is shown in Figure 2.4 which has zeros below the diagonal.

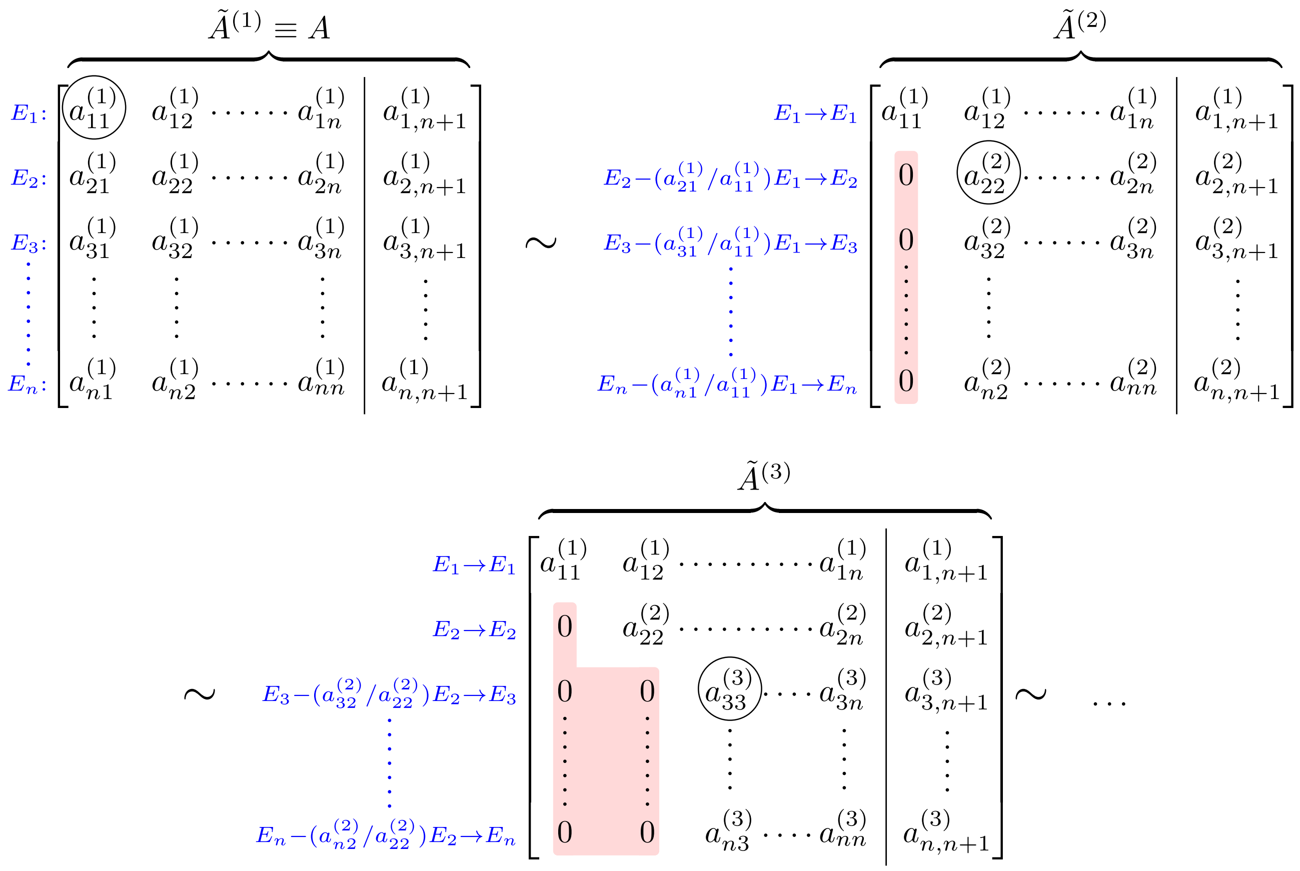

$\tilde{A}^{(1)} \equiv A$
$\tilde{A}^{(2)}$
$\tilde{A}^{(3)}$
E_1:
E_2:
E_3:
E_n:
$E_1 \to E_1$
$E_2 - (a_{21}^{(1)}/a_{11}^{(1)})E_1 \to E_2$
$E_3 - (a_{31}^{(1)}/a_{11}^{(1)})E_1 \to E_3$
$E_n - (a_{n1}^{(1)}/a_{11}^{(1)})E_1 \to E_n$
$E_1 \to E_1$
$E_2 \to E_2$
$E_3 - (a_{32}^{(2)}/a_{22}^{(2)})E_2 \to E_3$
$E_n - (a_{n2}^{(2)}/a_{22}^{(2)})E_2 \to E_n$

Figure 2.2: First steps of Gaussian Elimination

Recall that at the i-th step, we need to make zeros on the rows below E_i. Thus for $j = i+1, \dots, n$:

$$E_j \to E_j - (a_{ji}^{(i)}/a_{ii}^{(i)})E_i = \underbrace{\left[\overbrace{0,\dots,0}^{i-1}, a_{ji}^{(i)}, a_{j,i+1}^{(i)}, \dots a_{j,n+1}^{(i)}\right]}_{E_j} - (a_{ji}^{(i)}/a_{ii}^{(i)}) \underbrace{\left[\overbrace{0,\dots,0}^{i-1}, a_{ii}^{(i)}, a_{i,i+1}^{(i)}, \dots a_{i,n+1}^{(i)}\right]}_{E_i} \tag{2.6}$$

Hence the new coordinates for E_j are

$$E_j = \left[\overbrace{0,\dots,0}^{i-1}, \overbrace{a_{j,i}^{(i+1)}}^{=0}, a_{j,i+1}^{(i+1)}, a_{j,i+2}^{(i+1)}, \dots a_{j,n+1}^{(i+1)}\right] \tag{2.7}$$

where for $k = i+1, \dots, n+1$

$$a_{jk}^{(i+1)} = a_{jk}^{(i)} - (a_{ji}^{(i)}/a_{ii}^{(i)})a_{ik}^{(i)} \tag{2.8}$$

We summarise the Gaussian elimination (also called forward elimination) in Algorithm 2.1 . We have call it the "Naive" version because as you can see from Equation 2.8, the algorithm will not work if the pivot is zero at some step (i.e. if $a_{ii}^{(i)} = 0$ for some i) and if we naively do not do something about!. We will, of course address this potential issue later in Section 2.1.6.

Note

It is important to note that at each step of Gaussian elimination we re-write the entries of the matrix $\tilde{A}$, i.e. we do not store all the intermediate matrices $\tilde{A}^{(1)}, \dots \tilde{A}^{(n)}$. So the memory requirements of the algorithms is the same as the inputs.

The python code from Listing 2.1 implements Gaussian elimination as a function which will be later tested in Important 12 with the system from Section 2.1.1.

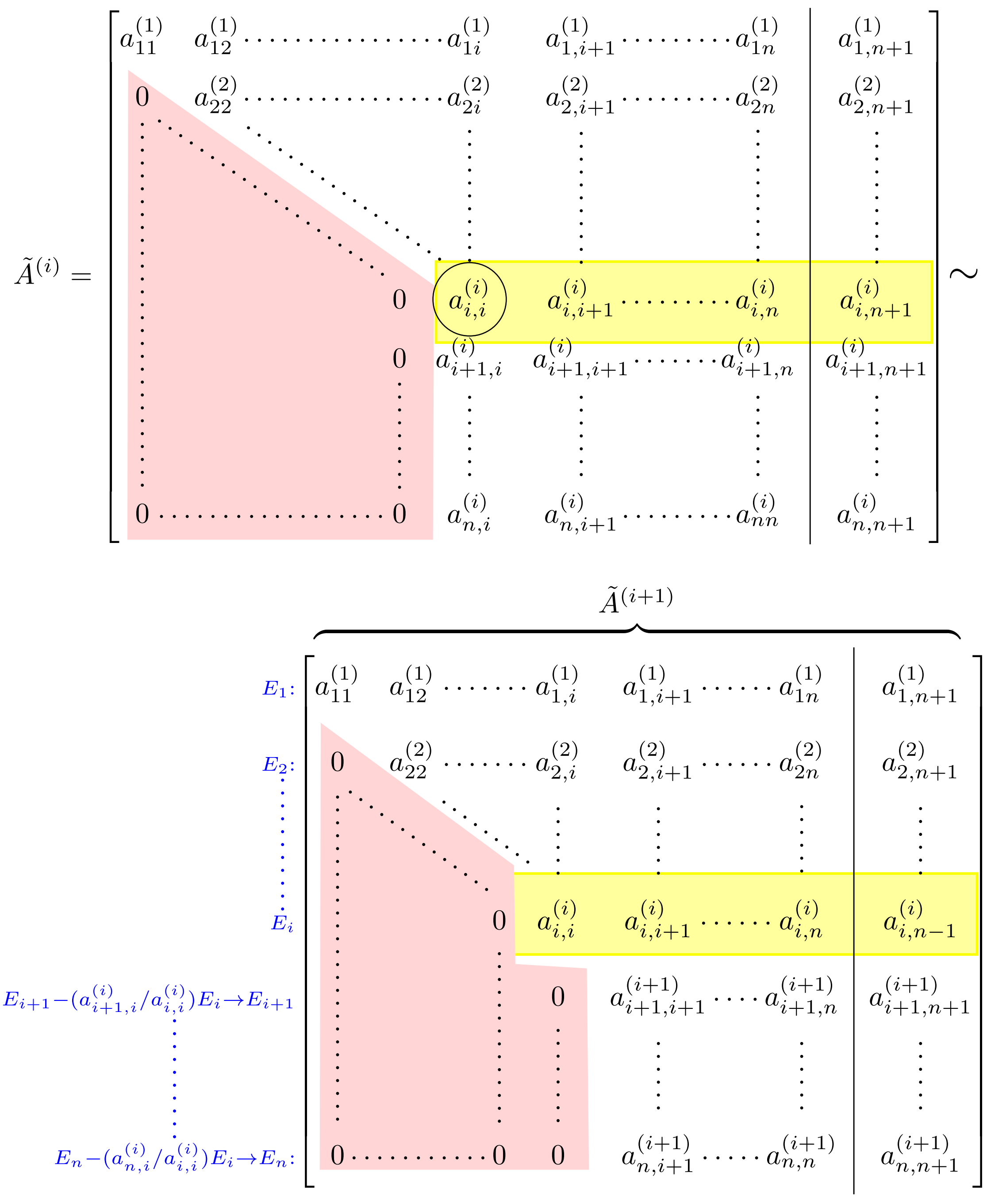

$\tilde{A}^{(i)} =$
$\sim$
$\tilde{A}^{(i+1)}$
E_1:
E_2:
E_i
$E_{i+1}-(a^{(i)}_{i+1,i}/a^{(i)}_{i,i})E_i \to E_{i+1}$
$E_n-(a^{(i)}_{n,i}/a^{(i)}_{i,i})E_i \to E_n$:

Figure 2.3: ith-step of Gaussian Elimination

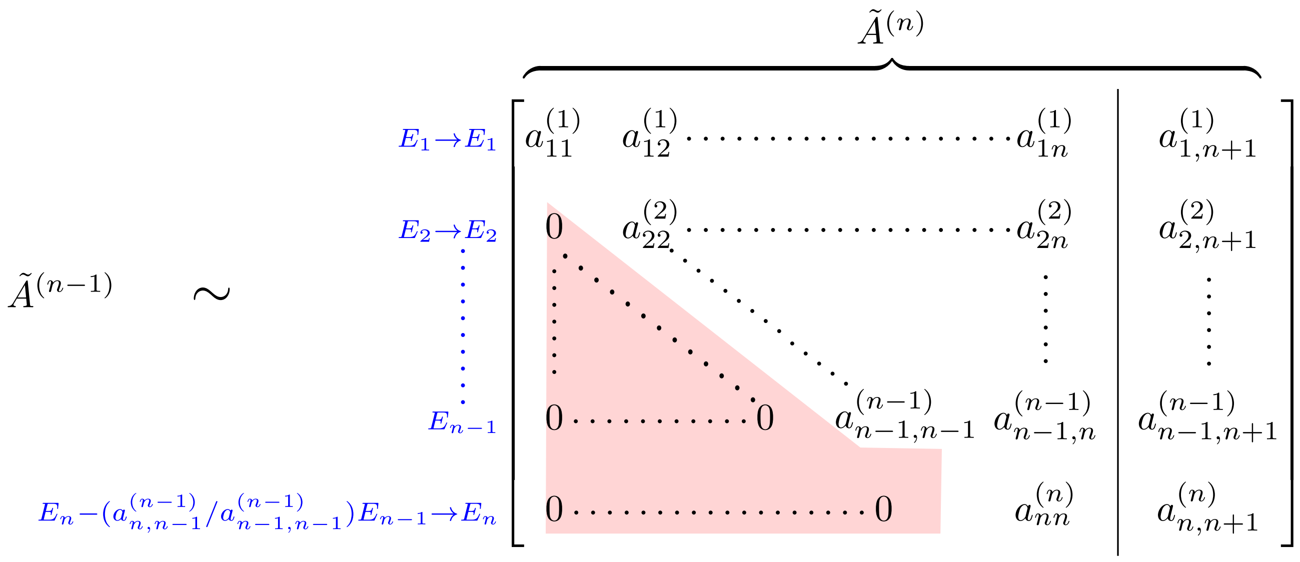

$\tilde{A}^{(n)}$
$\tilde{A}^{(n-1)} \sim$
$E_1 \to E_1$
$E_2 \to E_2$
E_{n-1}
$E_n - (a^{(n-1)}_{n,n-1}/a^{(n-1)}_{n-1,n-1})E_{n-1} \to E_n$
$a^{(1)}_{11}$ $a^{(1)}_{12}$ $a^{(1)}_{1n}$ $a^{(1)}_{1,n+1}$
0 $a^{(2)}_{22}$ $a^{(2)}_{2n}$ $a^{(2)}_{2,n+1}$
0 0 $a^{(n-1)}_{n-1,n-1}$ $a^{(n-1)}_{n-1,n}$ $a^{(n-1)}_{n-1,n+1}$
0 0 $a^{(n)}_{nn}$ $a^{(n)}_{n,n+1}$

Figure 2.4: Last step of Gaussian Elimination

Algorithm 2.1 Naive Gaussian Elimination

```
1: **Input:** $n \times n$ matrix $A$. $n \times 1$ vector $b$.
2: **Output:** $\tilde{A}$ in reduced form.
3: **procedure** NaiveGaussianElimination($A$,$b$)
4:     Define augmented matrix $\tilde{A} = [A\,|\,b]$. Denote by $a_{ij}$ the entries of $\tilde{A}$.
5:     **for** $i = 1$ **to** $n-1$ **do**
6:         **for** $j = i+1$ **to** $n$ **do**
7:             Compute $m_{ji} = a_{ji}/a_{ii}$
8:             Set $a_{ji} = 0$
9:             **for** $k = i+1$ **to** $n+1$ **do**
10:                $a_{jk} = a_{jk} - m_{ji}a_{ik}$
11:            **end for**
12:        **end for**
13:    **end for**
14: **end procedure**
```

2.1.3 Backward substitution

Once we have produced matrix $\tilde{A}^{(n)}$, the original system (see Equation 2.1) is reduced to

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + \cdots + a_{1n}x_n &= a_{1,n+1} \\ a_{22}x_2 + a_{23}x_3 + \cdots + \cdots + a_{2n}x_n &= a_{2,n+1} \\ \vdots &= \vdots \\ a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n &= a_{n-1,n+1} \\ a_{nn}x_n &= a_{n,n+1} \end{aligned} \tag{2.9}$$

where for ease in the notation we have omitted the superscripts $^{(1)}, \ldots ^{(n)}$, but we emphasize that these coefficients are different from those of the original input matrix $\tilde{A}$.

As before we can use back substitution. From the nth equation we solve for x_n:

$$x_n = \frac{a_{n,n+1}}{a_{nn}}. \tag{2.10}$$

From the $(n-1)$ equation we have

$$x_{n-1} = \frac{a_{n-1,n+1} - a_{n-1,n}x_n}{a_{n-1,n-1}}$$

and continuing this procedure we have

Listing 2.1 Implementation of Naive Gaussian elimination

```python
import numpy as np

def naive_gaussian_elimination(A,b):
    """
    Returns numpy array with the augmented matrix tildeA obtained by
    Gaussian elimination starting from the augmented matrix [A b].

    Parameters (Inputs)
    ----------
    A : numpy.ndarray of shape (n,n)
        Array representing the square matrix A.
    b : numpy.ndarray of shape (n,1)
        Array representing the column vector b.

    Returns
    -------
    tildeA : numpy.ndarray of shape (n,n+1)
        Array representing the augmented matrix tildeA.
    """

    # Create the initial augmented matrix
    tildeA=np.hstack((A,b))

    #number of rows:
    n = np.shape(tildeA)[0]

    # Start Gaussian Elimination
    for i in range(n-1):
        for j in np.arange(i+1,n):
            #compute multiplier
            m = tildeA[j,i] / tildeA[i,i]
            #make zeros below pivot
            tildeA[j,i] = 0
            #update jth row (only the non-zero elements)
            for k in np.arange(i+1,n+1):
                tildeA[j,k] = tildeA[j,k] - m * tildeA[i,k]


    return tildeA
```

$$x_i = \frac{a_{i,n+1} - \sum\limits_{j=i+1}^{n} a_{ij}x_j}{a_{ii}}$$

for $i = n-1, n-2 \dots 2, 1$.

Notice that the row swapping discussed earlier does not prevent from potentially having $a_{nn} = 0$. In this case x_n in Equation 2.10 is not defined and thus the system does not have a unique solution.

The pseudocode is given in Algorithm 2.2 where, for ease in the notation, we omit the superindex for the entries of the reduced-form matrix $\tilde{A}$.

In Listing 2.2 we implement a python function for backward substitution. In Important 12 we then combine this function with the one for Gaussian elimination implemented in Listing 2.1 to solve the system from Section 2.1.1.

Algorithm 2.2 Backward Substitution

```
1: Input: $n \times (n+1)$ augmented matrix $\tilde{A}$.
2: Output: $\mathbf{x}$ solution to $A\mathbf{x} = \mathbf{b}$.
3: $\tilde{A}$ is in reduced form (entries denoted by $a_{ij}$).
4: procedure BackSubstitution($\tilde{A}$)
5:     if $a_{nn} = 0$ then
6:         Terminate.
7:         Output: the system does not have unique solution
8:     end if
9:     Set $x_n = a_{n,n+1}/a_{nn}$
10:    for $i = n-1$ to 1 do
11:        Compute $s = \sum_{j=i+1}^{n} a_{ij}x_j$
12:        $x_i = (a_{i,n+1} - s)/a_{ii}$
13:    end for
14: end procedure
```

❗ Important 12: Computing Activity

Create a file called `directsolvers.py` and include in this file the functions `naive_gaussian_elimination` from Listing 2.1 and the function `backward_substitution` from Listing 2.2. Copy the code below into a file `main_direct.py` which calls `direct_solvers.py` as module to test the algorithms.

Listing 2.3 `main_direct.py`

```python
import directsolvers as dsolve
import numpy

A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)
b=np.array([[2],[6],[12]], dtype=float)

tildeA=    dsolve.naive_gaussian_elimination(A, b)
x=  dsolve. backward_substitution(tildeA)
print(f"Reduced matrix:\ntildeA =\n{tildeA}")
print(f"solution to the linear system\nx =\n{x}")
```

```
Reduced matrix:
tildeA =
[[2. 3. 4. 2.]
 [0. 3. 2. 2.]
 [0. 0. 4. 2.]]
solution to the linear system
x =
[[-0.5       ]
 [ 0.33333333]
 [ 0.5       ]]
```

2.1.4 Complexity (Operation Counts)

The complexity or computational cost of direct methods for solving linear systems is measured in terms of the number of elementary operations required to complete the algorithm. The more operations we have, the longer the time the code will take to complete. In the code from Listing 2.4 we compute the execution time (or CPU time) of additions, subtractions, multiplications, divisions and comparisons. In this code we also show how Python's module `time` can be used to compute execution time.

Let us now count the operations for the Gaussian elimination algorithm shown in Algorithm 2.1 . We will focus first on multiplications and divisions. Consider the ith step of the outer loop and the jth step of the first inner

Listing 2.2 Implementation of Backward substitution

```
def backward_substitution(tildeA):
    """
    Returns an array representing the solution x of Ax=b computed using
    backward substitution after the augmented matrix tildeA has been sucesfully
 ↪  computed
    via Gaussian elimination.

    Parameters
    ----------
    tildeA : numpy.ndarray of shape (n,n+1)
        Array representing the augmented matrix [U v].
    n : int
        Integer that is at least 2.

    Returns
    -------
    x : numpy.ndarray of shape (n,1)
        Array representing the solution x.
    """
    n = np.shape(tildeA)[0]
    x=np.zeros([n,1])

    #check for a_{n,n} =0
    if tildeA[n-1, n-1] == 0:
      raise ValueError("The system does not have unique solution.")

    #start back substitution
    x[n-1] = tildeA[n-1,n] / tildeA[n-1,n-1]

    for i in np.arange(n-1,0,-1):
        #Computing the sum
        s = 0
        for j in np.arange(i+1,n+1):
            s = s + tildeA[i-1,j-1]*x[j-1]
        x[i-1] = (tildeA[i-1,n] - s) / tildeA[i-1,i-1]

    return x
```

loop. Note that line 7 involves 1 division to compute the multiplier m_{ij}. Then, we compute (line 10)

$$a_{jk} = a_{jk} - m_{ji}\, a_{ik} \tag{2.11}$$

which, for each iteration k, requires 1 multiplication. Since k goes from $k = i+1$ to $k = n+1$ the total number of multiplications at the ith and jth step is $n+1-(i+1)+1 = n-i+1$. Therefore, there are

$$\overset{div.}{\widehat{1}} + \overbrace{n+1-i}^{mult.} = 2+n-i$$

multiplications and divisions at the ith and jth step of Algorithm 2.1 .

To account for the multiplications and divisions for all $j = i+1, \ldots n$ at the ith step of the outer loop we note that the above count is done $n-(i+1)+1 = (n-i)$ times and thus:

$$\overbrace{(n-i)}^{\substack{from\ counting\\ j=i+1,\ldots n}} \quad (2+n-i).$$

We now add over the index i, which goes from $i = 1$ to $n-1$, to find that the total number of multiplications and divisions is:

$$\begin{aligned}
\mathcal{C}_{mul/div}(n) = \sum_{i=1}^{n-1}(n-i)(n-i+2) &= \overbrace{\sum_{i=1}^{n-1}(n-i)^2 + 2\sum_{i=1}^{n-1}(n-i) = \sum_{m=1}^{n-1} m^2 + 2\sum_{m=1}^{n-1} m}^{relabel \quad m=n-i} \\
&= \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2} \qquad (from\ Lemma's) \\
&= \frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n
\end{aligned}$$

where we have used lemmas from calculus that can be found in Lemma A.1.

From the previous expression we observe that for large n the dominant term is $n^3/3$. In fact we can make this more precise by noticing that, since $n > 1$,

$$\frac{1}{2}n^2 - \frac{5}{6}n \le \frac{1}{2}n^2 \le \frac{1}{2}n^3$$

Hence:

$$\mathcal{C}_{mul/div}(n) = \frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n \le \frac{1}{3}n^3 + \frac{1}{2}n^2 = \frac{5}{6}n^3$$

From the definition of Big-O notation (see Definition A.2) we can simply write

$$\mathcal{C}_{mul/div}(n) = \mathcal{O}(n^3)$$

as $n \to \infty$.

Using similar arguments, in Exercise 2.5 you will show that the number of additions/subtractions is

$$\mathcal{C}_{add/sub}(n) = \sum_{i=1}^{n-1}(n-i)(n-i+1) = \frac{1}{3}n^3 - \frac{1}{3}n = \mathcal{O}(n^3)$$

Combining the results above we have that the total number of elementary operations is

$$\mathcal{C}_{total}(n) = \mathcal{C}_{add/sub}(n) + \mathcal{C}_{mul/div}(n) \le Cn^3 \Longrightarrow \mathcal{C}_{total}(n) = \mathcal{O}(n^3) \qquad \text{as } n \to \infty$$

Note

It is worth saying that when we have results of the form $f(n) = \mathcal{O}(n^p)$ it is implicitly assumed that p is the smallest integer for which the big-O notation is valid. For the case above we have $\mathcal{C}_{total}(n) \le Cn^3$ but note that

$$\mathcal{C}_{total}(n) \le Cn^3 \le Cn^4 \le Cn^5, \ldots$$

so, while we also have $\mathcal{C}_{total}(n) = \mathcal{O}(n^p)$ for $p \ge 3$, it is important to note that $p = 3$ is the smallest (optimal) value for which the big-O notation can be applied.

Since we are interested in the case of large n, we do not focus on the constant C (although we have calculated it). If we want to verify this result we can conduct similar to the analysis of convergence rate from the previous chapter. Namely, taking $\log_{10}$ from both sides from $\mathcal{C}_{total}(n) \le Cn^3$ yields:

$$\log_{10}(\mathcal{C}_{total}(n)) \le \log_{10}(C) + 3\log_{10}(n)$$

Therefore, to verify this result, plotting $\mathcal{C}_{total}(n)$ vs n on a log-log scale should display a straight line with slope 3.

Therefore, we have that the cost of the computations increases with the cubic power of n. Thus, if we were to increase the size of the matrix A (from the original system) by a factor of 10, the number of operations will increase by a factor of 1000.

In Listing 2.5 we display python code to test these results by increasing the size of a random linear system. However, instead of counting the number of operations, we use the CPU time i.e. the time it takes to compute Gaussian elimination code. The execution time is, of course, proportional to the number of operations and so it is a good proxy to understand the behaviour of $\mathcal{C}_{total}(n)$. From Figure 2.5 we show $\mathcal{C}_{total}(n)$ vs n on a log-log scale where we also include plots of lines with slopes 3 and 2. This figure confirms that $\mathcal{C}_{total}(n)$ behaves like n^3.

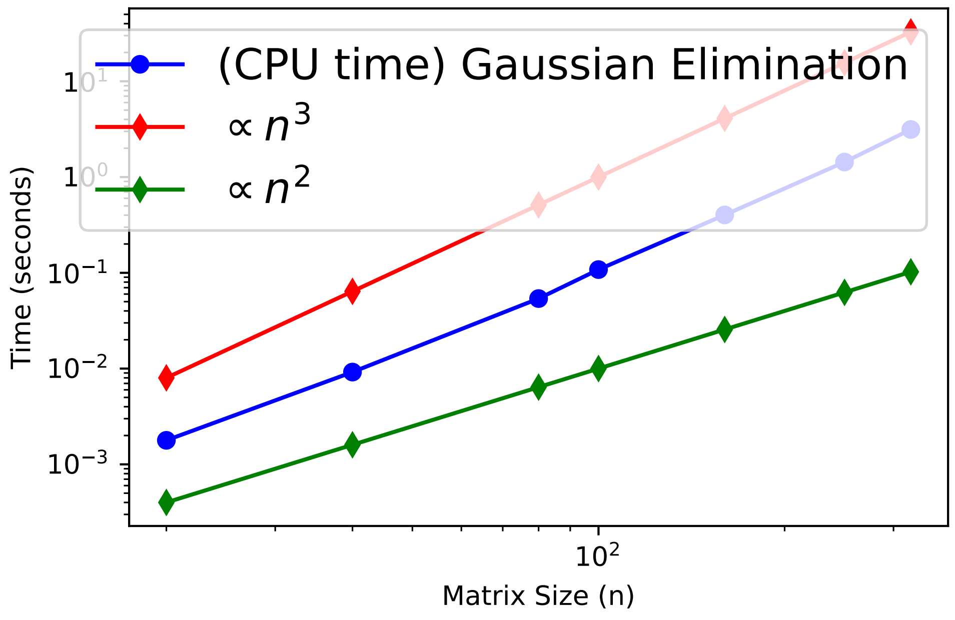


Figure 2.5: Asymptotic behaviour of computation time

! Important 13: Computer Activity

Use the code from Listing 2.5 to reproduce Figure 2.5 (or something similar perhaps?)

i Note

In Exercise 2.6 you will show that the complexity of backward substitution is $\mathcal{O}(n^2)$. Therefore, the complexity of Gaussian elimination with backward substitution to solve a linear system of size n is $\mathcal{O}(n^3)$.

2.1.5 Vectorisation

Implementation style may have a substantial effect on the performance of algorithms. In lines 35-36 of Listing 2.1, for example, we have used a `for` loop to update each of the entries of E_j (see line 10 in Algorithm 2.1). We can, however, replace

```python
for k in np.arange(i+1,n+1):
  tildeA[j,k] = tildeA[j,k] - m * tildeA[i,k]
```

by

```python
tildeA[j, i+1:] = tildeA[j, i+1:]- m * tildeA[i, i+1:]
```

or

```python
tildeA[j, i+1:] -= m * tildeA[i, i+1:]
```

Instead of iterating over each non-zero entry of E_j, we update the entire row with a single instruction. This approach is an example of **vectorisation**, which aims to minimise the number of instructions issued by the code. Although the total number of operations remains unchanged, vectorisation can significantly reduce execution time. The code from Listing 2.6 is similar to Listing 2.5 but it uses a function `gaussian_elimination_vectorised(A, b)` where we have implemented the above vectorisation to Listing 2.7 as described above. Although the operation count is still the same ($\mathcal{O}(n^3)$), the execution time has been reduced substantially.

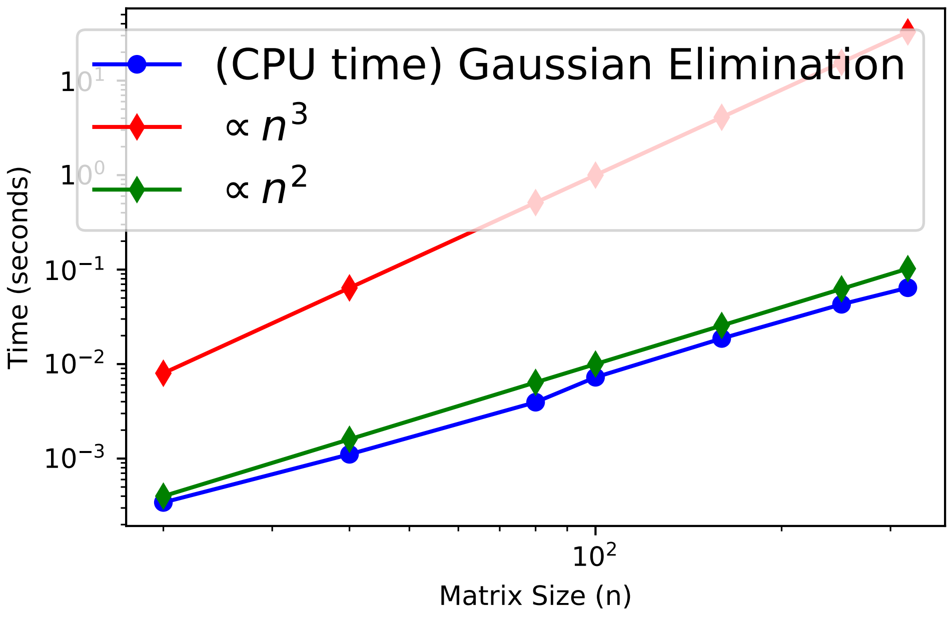


Figure 2.6: Asymptotic behaviour of computation time

! Important 14: Computer Activity

Write a function `gaussian_elimination_vectorised` using the vectorisation discussed earlier. Add your function to `directsolvers` and use the code from Listing 2.6 to test your function and to produce a plot similar to that in Figure 2.6.

2.1.6 Standard Pivoting

Note when we loop over the index i (row index) the algorithm is not well defined if the pivot is zero, i.e. $a_{ii} = 0$. However, if $a_{ii} = 0$ for any of the $i = 1, \dots, n-1$ we can simply swap row E_i with any of row below E_j (below E_i) that has non-zero entry a_{ji}. Since we want to do this systematically, we can simply choose p to the smallest integer p with $i \le p \le n$ and $a_{pi} \ne 0$ and then swap E_i and E_p (denoted by $E_p \leftrightarrow E_i$). Notice that if no such p can be found, then the linear system does not have a unique solution. This procedure is called **standard pivoting** which we combine with Gaussian elimination in Algorithm 2.3 .

Note 4: Note

A very important results is that a square matrix A is non-singular if Gaussian elimination (and backward substitution) with row interchanges can be performed on the system $A\mathbf{x} = \mathbf{b}$ for any $\mathbf{b} \in \mathbb{R}^n$.

Exercise 2.1. Apply Gaussian elimination (with standard pivoting) with back substitution to solve the system $A\mathbf{x} = \mathbf{b}$ where

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 4 \end{bmatrix}.$$

Solution

The augmented matrix is:

$$\tilde{A} = [A \,|\, \mathbf{b}] = \left[\begin{array}{ccc|c} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{array}\right]$$

Since $a_{11} = 0$, we swap with the smallest p such that $a_{p1} \ne 0$. Note that $a_{21} = 1 = a_{31}$ so we select a_{21} and swap $E_1 \leftrightarrow E_2$ so

$$\tilde{A} \sim \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{array}\right]$$

Now we do $E_2 \to E_2 - (0/1)E_1 = E_2$ (so E_2 remains unchanged) and $E_3 \to E_3 - (1/1)E_1 = [0\ 1\ 0\ 3]$. Thus

$$\tilde{A} \sim \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 3 \end{array}\right]$$

In the last step we conduct $E_3 \to E_3 - (1/1)E_2 = [0\ 0\ -1\ 2]$. Hence

$$\tilde{A} \sim \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 3 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & -1 & 2 \end{array}\right]$$

Back substitution:

$$\begin{aligned} x_3 &= 2/(-1) = -2 \\ x_2 &= 1 - x_3 = 1 - (-2) = 3 \\ x_1 &= 1 - x_2 - x_3 = 1 - 3 - (-2) = 0 \end{aligned}$$

Hence $\mathbf{x} = \begin{bmatrix} 0 \\ 3 \\ -2 \end{bmatrix}$.

Note 5: Swapping rows (and columns) in python

Consider the following numpy array

```
import numpy as np

A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)
```

Note that the first row can be access via

```
print(A[0])
```

```
[2. 3. 4.]
```

which is equivalent to slicing:

```
print(A[0,:])
```

```
[2. 3. 4.]
```

It should be obvious that if we do

```
A[0]=A[1]
A[1]=A[0]
print(A)
```

```
[[ 4.   9. 10.]
 [ 4.   9. 10.]
 [ 6. 15. 20.]]
```

we are only moving the second row into the first not chancing the second row as this was first changed. One can be tempted to do

```
A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)

temp=A[0]
A[0]=A[1]
A[1]=temp
print(A)
```

```
[[ 4.   9. 10.]
```

```
 [ 4.  9. 10.]
 [ 6. 15. 20.]]
```
which is not what we want. Notice that A is an object, when we define the new variable `temp=A[0]` we are simply accessing a view to A and if we change A we change `temp` (and viceversa). Instead we need to use the `.copy()` method of numpy arrays. For example:

```
A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)

temp=A[0].copy()  #this is a new variable
A[0]=A[1]
A[1]=temp
```

We can also use:

```
A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)

(A[0], A[1])=(A[1].copy(), A[0].copy())
print(A)
```

```
[[ 4.  9. 10.]
 [ 2.  3.  4.]
 [ 6. 15. 20.]]
```
Alternatively, we can employ *advanced indexing* by which we access the first and second row of A via `A[ [0, 1] ]`:

```
print(A[ [0, 1] ])
```

```
[[ 4.  9. 10.]
 [ 2.  3.  4.]]
```
note that we are passing the list `[0, 1]` to `A[]`. The nice thing about advanced indexing is that it returns a copy. So to swap rows we can do

```
 A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)
A[ [0, 1] ] =A [[ 1, 0 ] ]
print(A)
```

```
[[ 4.  9. 10.]
 [ 2.  3.  4.]
 [ 6. 15. 20.]]
```
Note that since `A[ [0, 1] ]` is passing the list `[0, 1]` to `A[]`, it acts along the first dimension of `A` (i.e. rows). In other words, `A[ [0, 1] ]` is the same as `A[ [0, 1],: ]`. If we wanted to swap the rows but only up the first two columns we need something like

```
 A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)
A[ [0, 1], 0:1] =A [[ 1, 0 ] ,0:1]
print(A)
```

```
[[ 4.  3.  4.]
 [ 2.  9. 10.]
 [ 6. 15. 20.]]
```
In addition, if instead of swapping rows we want to swap columns, say column 2 and 3 we pass the lists [1,2] and [2,1] to the second dimension of `A`:

```
 A=np.array([[2,3,4],[4,9,10],[6,15,20]], dtype=float)
A[:, [1, 2]] =A [:, [2,1]]
print(A)
```

```
[[ 2.  4.  3.]
 [ 4. 10.  9.]
 [ 6. 20. 15.]]
```

The python code from Listing 2.7 implements Gaussian elimination with standard pivoting as a function which is then tested with the system from Exercise 2.1. Note that we have used advanced indexing to swap rows as discussed in Note 5.

Let us now discussed the complexity of Algorithm 2.3 . While the number of arithmetic operations is the sames is that of Algorithm 2.1 , the additional pivoting step only involves comparisons. As shown in Section 2.1.4, the computational cost of comparisons is similar to those of of arithmetic operations.

We first note that at each *ith* step ($i = 1$ to $i = n-1$) of Gaussian elimination with standard pivoting we conduct 1 comparison to check whether $a_{ii} = 0$. Then, to find the smallest p such that $a_{pi} \neq 0$, one approach as that taken in Listing 2.7 is to loop for $p = i+1$ to $p = n$ to find whether $a_{pi} \neq 0$ which involves a maximum of $n - (i+1) + 1 = (n-i)$ comparisons. Note that the actual number of comparisons here could be less (if $a_{pi} \neq 0$ for $p < n$), but here we consider the worst-case scenario. Therefore, we have that at the i-th step there are $1 + (n-i)$ comparisons, and since $i = 1$ to $i = n-1$, using the same arguments as above, we have that the total number of comparisons is

$$\begin{aligned}\mathcal{C}_{com}(n) &= \sum_{i=1}^{n-1} 1 + (n-i)\\ &= \sum_{i=1}^{n-1} 1 + \sum_{i=1}^{n-1}(n-i) = (n-1) + \sum_{i=1}^{n-1} m\\ &= (n-1) + \frac{(n-1)n}{2} = \frac{n^2}{2} + \frac{n}{2} - 1\end{aligned}$$

Hence we have that $\mathcal{C}_{com}(n) = \mathcal{O}(n^2)$ and so for large n, the complexity of standard pivoting is smaller than the cost of arithmetic operations.

Combining the results above we have that the total number of elementary operations is

$$\mathcal{C}_{total}(n) = \mathcal{C}_{add/sub}(n) + \mathcal{C}_{mul/div}(n) + \mathcal{C}_{com}(n) \leq Cn^3 \Longrightarrow \mathcal{C}_{total}(n) = \mathcal{O}(n^3) \qquad \text{as } n \to \infty$$

Therefore, the added $\mathcal{O}(n^2)$ complexity of standard pivoting does not change the overall $\mathcal{O}(n^3)$ of Gaussian elimination without pivoting as $n \to \infty$.

2.1.7 Additional Exercises

Exercise 2.2. Consider the augmented matrices:

1. $$\tilde{A} = \left[\begin{array}{ccc|c} 1 & -1 & 2 & -6\\ 1 & 1 & 1 & -2\\ 2 & 0 & 2 & -10\end{array}\right]$$

2. and $$\tilde{A} = \left[\begin{array}{ccc|c} 1 & -1 & 2 & \alpha\\ 1 & -1 & 1 & \beta\\ 2 & 0 & 2 & \gamma\end{array}\right]$$

for $\alpha, \beta, \gamma \in \mathbb{R}$.

Apply Gaussian elimination to $\tilde{A}$. Make sure to apply row interchanges if needed as discussed in Section 2.1.6.

Solution

1. First and second steps:

$$\tilde{A} = \left[\begin{array}{ccc|c} 1 & -1 & 2 & -6\\ 1 & 1 & 1 & -2\\ 2 & 0 & 2 & -10\end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & -1 & 2 & -6\\ 0 & 2 & -1 & 4\\ 0 & 2 & -2 & 2\end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & -1 & 2 & -6\\ 0 & 2 & -1 & 4\\ 0 & 0 & -1 & -2\end{array}\right]$$

2. In the first step we have

$$\tilde{A} = \left[\begin{array}{ccc|c} 1 & -1 & 2 & \alpha\\ 1 & -1 & 1 & \beta\\ 2 & 0 & 2 & \gamma\end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & -1 & 2 & -6\\ 0 & 0 & -1 & \beta-\alpha\\ 0 & 2 & -2 & \gamma-\alpha\end{array}\right]$$

and since the pivot $a_{22} = 0$ we need to swap $E_2 \leftrightarrow E_3$ so

$$\tilde{A} \sim \left[\begin{array}{ccc|c} 1 & -1 & 2 & -6 \\ 0 & 2 & -2 & \gamma - \alpha \\ 0 & 0 & -1 & \beta - \alpha \end{array}\right]$$

Exercise 2.3. Use Gaussian elimination with backward substitution to solve the following linear systems, if possible and determine whether row interchanges are necessary

1.
$$\begin{aligned} x_1 - x_2 + 3x_3 &= \quad 2 \\ 3x_1 - 3x_2 + x_3 &= \quad -1 \\ x_1 + x_2 &= \quad 3 \end{aligned}$$

2.
$$\begin{aligned} 2x_1 - 1.5x_2 + 3x_3 &= \quad 1 \\ -x_1 + 2x_3 &= \quad 3 \\ 4x_1 - 4.5x_2 + 5x_3 &= \quad 1 \end{aligned}$$

3.
$$\begin{aligned} 2x_1 - 1.5x_2 + 3x_3 &= \quad 1 \\ -x_1 + 2x_2 &= \quad 3 \\ 4x_1 - 4.5x_2 + 5x_3 &= \quad 1 \end{aligned}$$

Solution

Please use hand calculations to practice the main concepts. You may use the codes from this section to verify your answers:

1.

```
A=np.array([[1,-1,3],[3,-3,1],[1,1,0]])
b=np.array([[2],[-1],[3]])
tildeA=   gaussian_elimination(A, b)
x=backward_substitution(tildeA)
print(x)
```

```
zero pivot found, swapping row  2  with row  3
[[1.1875]
 [1.8125]
 [0.875 ]]
```

2.

```
A=np.array([[2,-1.5,3],[-1,0,2],[4,4.5,5]])
b=np.array([[1],[3],[1]])
tildeA=   gaussian_elimination(A, b)
x=backward_substitution(tildeA)
print(x)
```

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

3.

```
raise ValueError("The system does not have unique solution.")

ValueError: The system does not have unique solution.
```

Exercise 2.4. Consider the system of Exercise 2.1 but instead of A consider the following matrix

$$A = \begin{bmatrix} \epsilon & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

with $0 < \epsilon \ll 1$. Apply Gaussian elimination with back substitution to show that the solution of the reduced system is the same as that of Exercise 2.1.

Solution

The augmented matrix is

$$\tilde{A} = \left[\begin{array}{ccc|c} \epsilon & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{array}\right]$$

In the first step we leave E_1 unchanged while we replace E_2 and E_3 as follows:

$$E_2 \to \overbrace{[1,1,1,1]}^{E_2} - \epsilon^{-1}\overbrace{[\epsilon,1,1,1]}^{E_1} = [0,(1-\epsilon^{-1}),(1-\epsilon^{-1}),(1-\epsilon^{-1})]$$
$$E_3 \to \overbrace{[1,2,1,4]}^{E_3} - \epsilon^{-1}\overbrace{[\epsilon,1,1,1]}^{E_1} = [0,(2-\epsilon^{-1}),(1-\epsilon^{-1}),(4-\epsilon^{-1})]$$

So

$$\tilde{A} = \begin{bmatrix} \epsilon & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \sim \begin{bmatrix} \epsilon & 1 & 1 & 1 \\ 0 & (1-\epsilon^{-1}) & (1-\epsilon^{-1}) & (1-\epsilon^{-1}) \\ 0 & (2-\epsilon^{-1}) & (1-\epsilon^{-1}) & (4-\epsilon^{-1}) \end{bmatrix}$$

In the second step we replace E_3 by

$$\overbrace{[0,(2-\epsilon^{-1}),(1-\epsilon^{-1}),(4-\epsilon^{-1})]}^{E_3} - \frac{(2-\epsilon^{-1})}{(1-\epsilon^{-1})}\overbrace{[0,(1-\epsilon^{-1}),(1-\epsilon^{-1}),(1-\epsilon^{-1})]}^{E_2}$$
$$= \Big[0,\, 0,\, (1-\epsilon^{-1}) - (2-\epsilon^{-1}), (4-\epsilon^{-1}) - (2-\epsilon^{-1})\Big]$$
$$= [0,0,-1,2]$$

Thus

$$\tilde{A} \sim \begin{bmatrix} \epsilon & 1 & 1 & 1 \\ 0 & (1-\epsilon^{-1}) & (1-\epsilon^{-1}) & (1-\epsilon^{-1}) \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

Then using back substitution

$$x_3 = -2$$
$$x_2 = \frac{1}{(1-\epsilon^{-1})}\Big[(1-\epsilon^{-1}) - (1-\epsilon^{-1})x_2\Big] = 3$$
$$x_1 = \frac{1}{\epsilon}(1 - x_3 - x_2) = 0$$

Exercise 2.5. Compute the total number of additions/subtractions in the Gaussian elimination method.

Solution

From Equation 2.11 we notice that at the ith and jth step there are $(1+n-i)$ subtractions since k goes from $k = i+1$ to $k = n+1$. Hence at the i-th step there are

$$\overbrace{(n-i)}^{\substack{countingfrom\\ j=i+1,\dots n}} \quad (n-i+1)$$

subtractions. Hence, the total number is (adding from $i = 1$ to $i = n-1$):

$$\begin{aligned}\mathcal{C}_{add/sub}(n) = \sum_{i=1}^{n-1}(n-i)(n-i+1) &= \overbrace{\sum_{i=1}^{n-1}(n-i)^2 + \sum_{i=1}^{n-1}(n-i) = \sum_{m=1}^{n-1} m^2 + \sum_{m=1}^{n-1} m}^{relabel\quad m=n-i} \\ &= \frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} \qquad (from\ Lemma's) \\ &= \frac{1}{3}n^3 - \frac{n}{3}\end{aligned}$$

where we have used lemmas from calculus that can be found in Lemma A.1.

Exercise 2.6. How many arithmetic operations are needed for backward substitution?

Solution

Multiplications/divisions: From line 8 of Algorithm 2.2 we require 1 division. At the ith step (line 10) we have ,

$$s = \sum_{j=i+1}^{n} a_{ij}x_j \tag{2.12}$$

which requires $n-(i+1)+1 = n-i$ multiplications, while for line 11 we have

$$x_i = (a_{i,n+1} - s)/a_{ii} \tag{2.13}$$

i.e. 1 division. Thus the total number of multiplications/divisions (since $i = 1\dots, n-1$) is

$$1 + \sum_{i=1}^{n-1}(n-i+1) = 1 + \sum_{i=1}^{n-1}(n-i) + \sum_{i=1}^{n-1} 1 = 1 + \sum_{i=1}^{n-1}(n-i) + n - 1 = n + \sum_{m=1}^{n-1} m$$

where we have relabeled $m = n-i$. Then, using one the results from Lemma A.1, the total number of multiplications/divisions is

$$n + \sum_{m=1}^{n-1} m = n + \frac{(n-1)n}{2} = \frac{n^2}{2} + \frac{n}{2}$$

Similarly from Equation 2.12 we have $n-(i+1) = n-i-1$ additions while from Equation 2.13 we have 1 subtraction. Hence the total number of addition/subtractions (since $i = 1\dots, n-1$) is

$$\sum_{i=1}^{n-1}(n-(i+1)+1) = \sum_{i=1}^{n-1}(n-i) = \sum_{m=1}^{n-1} m = \frac{(n-1)n}{2} + n = \frac{n^2}{2} - \frac{n}{2}$$

Note that the total number of operations is $\mathcal{O}(n^2)$.

2.1.8 Additional Computing Activities

! Important 15: Computing Activity

The aim of this activity is for you to verify that the complexity of backsubstitution is $\mathcal{O}(n^2)$ (see Exercise 2.6).

Write a code similar to the one from Listing 2.5, but instead of computing CPU time for `naive_gaussian_elimination`, compute the time of your function `backward_substitution` and plot it versus the size of the linear system n. Note that to do this you have to first call `naive_gaussian_elimination` to produce a reduced matrix before using Gaussian elimination. Include in your plots the graphs of lines proportional to n^2 and n. You should obtain something similar to Figure 2.7.

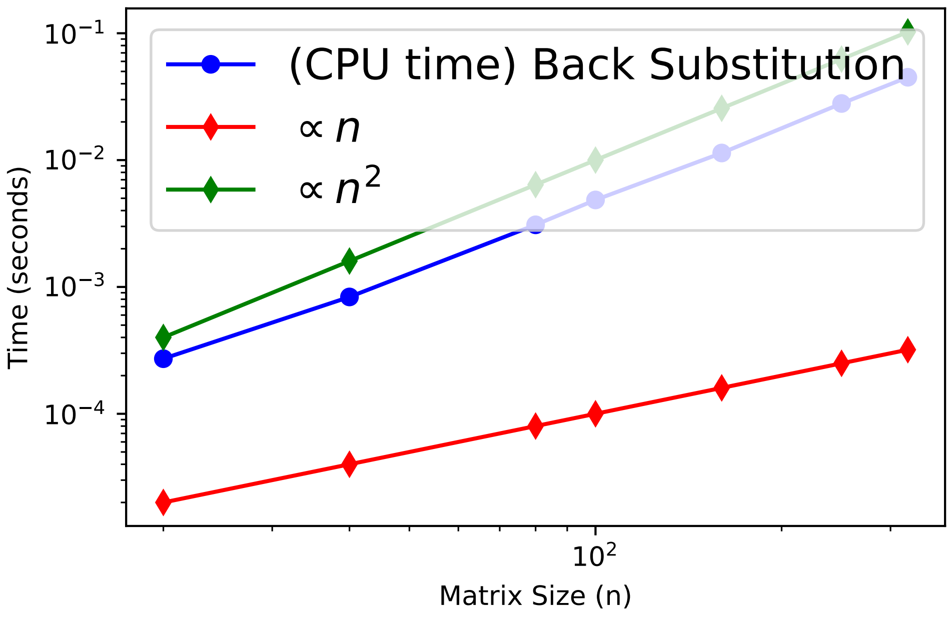


Figure 2.7: Asymptotic behaviour of computation time

! Important 16: Computing Activity

Note that in lines 31-33 of Listing 2.2 we computed $s = \sum_{j=i+1}^{n} a_{ij} x_j$ using a `for` loop. We can make this more efficient using vectorisation. Note that if we define $\mathbf{x}_{i+1} = (x_{i+1}, x_{i+2}, \dots, x_n)^t$ and $\mathbf{a}_{i,i+1} = (a_{i,i+1}, a_{i,i+2}, \dots, a_{i,n})^t$, then we can write s as the inner (or dot) product of these two vectors, i.e.

$$s = \sum_{j=i+1}^{n} a_{ij} x_j = \mathbf{x}_{i+1} \cdot \mathbf{a}_{i,i+1}$$

Therefore, instead of a the `for` loop we can take advantage of the vectorised python function `np.dot` where the summation is done in a vectorised fashion and hence more efficiently.
Have a go at writing a function `backward_substitution_vectorised` that uses dot product as described above instead of the `for` loop. Add your function to `directsolvers.py` and use it in the code from Important 15 instead of `backward_substitution`. You should obtain improved performance as shown in Figure 2.8.

2.2 Pivoting Strategies and round-off errors

2.2.1 Motivation

Consider the linear system $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 4 \end{bmatrix} \tag{2.14}$$

In Exercise 2.1 we showed that the exact solution to the linear system is

$$\mathbf{x} = \begin{bmatrix} 0 \\ 3 \\ -2 \end{bmatrix}. \tag{2.15}$$

Using the codes introduced in the previous section we may also verify that this solution is computed accurately and noted that code swapped E_1 and E_2 since the pivot element $a_{11} = 0$:

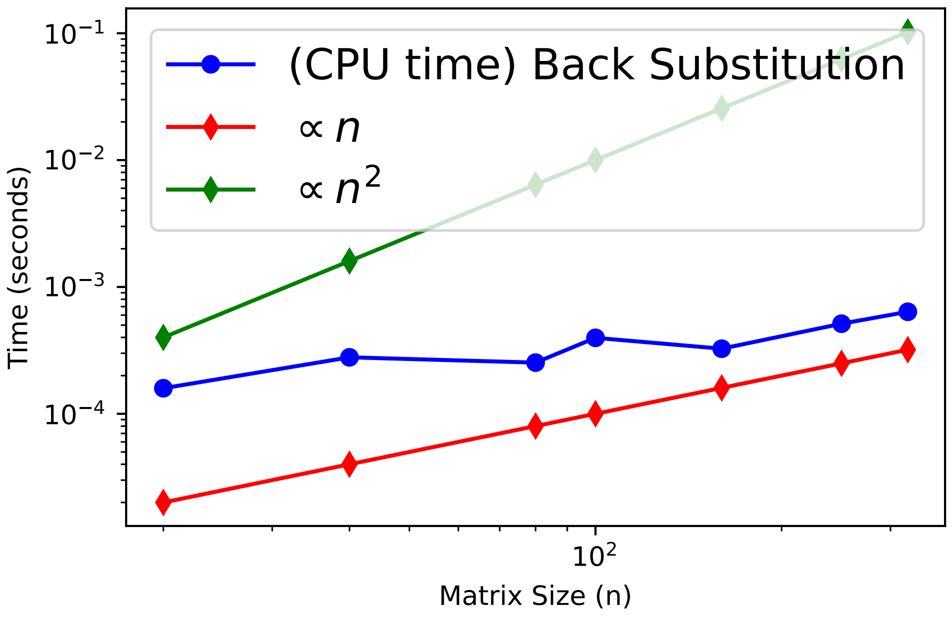


Figure 2.8: Asymptotic behaviour of computation time

```
A=np.array([[0,1.0,1.0],[1.0 , 1.0,1.0],[1.0,2.0,1.0]])
b=np.array([[1.0],[1.0],[4.0]])

tildeA=   gaussian_elimination(A, b)
x=backward_substitution(tildeA)
print(f"Reduced matrix:\ntildeA =\n{tildeA}")
print(f"solution to the linear system\nx =\n{x}")
```

```
zero pivot found, swapping row  1  with row  2
Reduced matrix:
tildeA =
[[ 1.  1.  1.  1.]
 [ 0.  1.  1.  1.]
 [ 0.  0. -1.  2.]]
solution to the linear system
x =
[[ 0.]
 [ 3.]
 [-2.]]
```

Consider now the same **b** but instead of matrix A from Equation 2.14 suppose we are given

$$A = \begin{bmatrix} \epsilon & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.16}$$

with $0 < \epsilon \ll 1$. So instead of $a_{11} = 0$ we have replaced this pivot element with a small number. In this case, you showed in Exercise 2.4 that the exact solution is also Equation 2.15 irrespective of the value of ϵ. However, if we use $\epsilon = 10^{-16}$ in our codes for Gaussian elimination and back substitution we obtain:

```
A=np.array([[1e-16,1.0,1.0],[1.0 , 1.0,1.0],[1.0,2.0,1.0]])
b=np.array([[1.0],[1.0],[4.0]])
tildeA=   gaussian_elimination(A, b)
x=backward_substitution(tildeA)
print(f"Reduced matrix:\ntildeA =\n{tildeA}")
print(f"solution to the linear system\nx =\n{x}")
```

```
Reduced matrix:
```

```
tildeA =
[[ 1.e-16  1.e+00  1.e+00  1.e+00]
 [ 0.e+00 -1.e+16 -1.e+16 -1.e+16]
 [ 0.e+00  0.e+00 -2.e+00  2.e+00]]
solution to the linear system
x =
[[ 0.]
 [ 2.]
 [-1.]]
```

which provides the wrong solution. This happens because of the numerical approximation that the computer makes of the numbers that we provided. This results in loss of accuracy that propagates in the computations. To understand this in a bit more detail we (very briefly) introduce some aspects of floating point arithmetic.

2.2.2 A brief note on floating point arithmetic

As you probably know, most computer platforms store numbers using a finite-digit binary representation of real numbers. This binary representation is a specific type of **floating-point system**, which is a finite set of real numbers designed to approximate $\mathbb{R}$. The elements of this system are referred to as floating points or floats. Most common binary systems can use 32 or 64 bits to store those numbers and the corresponding elements are referred to as single or double precision floats.

Because the set of floating points is finite, most real numbers cannot be represented exactly. Consequently, for any given $x \in \mathbb{R}$ the computer approximates x with a floating-point number, denoted as $\mathrm{fl}(x)$, which is the closest value to x according to certain rounding rules specified by the IEEE. The error resulting from this approximation is known as round-off error.

The error incurred when conducting such an approximation is called round-off error. For the binary representation, it can be shown that

$$\left|\frac{\mathrm{fl}(x) - x}{x}\right| \leq \epsilon_{mach} \tag{2.17}$$

where the term ϵ_{mach} is called machine precision and for double-precision (64 bits) can found in python via:

```python
import sys

print(sys.float_info.epsilon)
```

```
2.220446049250313e-16
```

If you would like to know more how this number is derived please read X.

Note for example that, if we define the variable `z=0.1`, the variable that we actually store is:

```python
z=0.1
print(f"{z:.55f}")
```

```
0.1000000000000000055511151231257827021181583404541015625
```

This approximation is made by the computer because 0.1 cannot be represented in binary format with finite number of digits. Although round-off error is small, this error propagates when computing further arithmetic operations and can become very large.

Floating point arithmetic refers to the operations performed by the computer on floats. If we have two real numbers $x, y \in \mathbb{R}$, these operations are defined by

$$\begin{aligned} x \oplus y &:= \mathrm{fl}(\mathrm{fl}(x) + \mathrm{fl}(y)) \qquad & x \otimes y &:= \mathrm{fl}(\mathrm{fl}(x) \times \mathrm{fl}(y)) \\ x \ominus y &:= \mathrm{fl}(\mathrm{fl}(x) - \mathrm{fl}(y)) \qquad & x \oslash y &:= \mathrm{fl}(\mathrm{fl}(x) / \mathrm{fl}(y)) \end{aligned}$$

For the addition $x \oplus y$, for example, we note that first x and y are approximated by $\mathrm{fl}(x)$ and $\mathrm{fl}(y)$ respectively. Since the sum of two floats is not necessarily a float, the result is then approximated to the nearest float in the system.

2.2.3 Round-off error and Gaussian elimination

In order to understand, what went wrong in our specific case, let us inspect the steps in Gaussian elimination in the example above for which the augmented matrix is

$$\tilde{A} = \left[\begin{array}{ccc|c} \epsilon & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{array}\right]$$

In the first step we leave E_1 unchanged while we replace E_2 and E_3 as follows:

$$E_2 \to \overbrace{[1,1,1,1]}^{E_2} - \epsilon^{-1} \overbrace{[\epsilon,1,1,1]}^{E_1} = [0,(1-\epsilon^{-1}),(1-\epsilon^{-1}),(1-\epsilon^{-1})]$$
$$E_3 \to \overbrace{[1,2,1,4]}^{E_3} - \epsilon^{-1} \overbrace{[\epsilon,1,1,1]}^{E_1} = [0,(2-\epsilon^{-1}),(1-\epsilon^{-1}),(4-\epsilon^{-1})]$$

So

$$\tilde{A} = \begin{bmatrix} \epsilon & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \sim \begin{bmatrix} \epsilon & 1 & 1 & 1 \\ 0 & (1-\epsilon^{-1}) & (1-\epsilon^{-1}) & (1-\epsilon^{-1}) \\ 0 & (2-\epsilon^{-1}) & (1-\epsilon^{-1}) & (4-\epsilon^{-1}) \end{bmatrix}$$

At this stage we note that we are subtracting numbers (i.e. 1 and $\epsilon^{-1} = 10^{16}$) of very different sizes. Note that

```python
eps=1e-16
z=1-1/eps
print(f"{z:.55f}")
```

```
-10000000000000000.0000000000000000000000000000000000000000000000000000000
```

and thus the stored value for $(1-\epsilon^{-1})$ is the same as the one for $-\epsilon^{-1}$:

```python
print(f"{-1/eps:.55f}")
```

```
-10000000000000000.0000000000000000000000000000000000000000000000000000000
```

We can verify this by

```python
print(1-1/eps == -1/eps)
```

```
True
```

The reason for this is that the closest float to $(1-\epsilon^{-1})$ is $-\epsilon^{-1}$. Obviously for such small ϵ and hence large ϵ^{-1}, approximating $(1-\epsilon^{-1})$ with $-\epsilon^{-1}$ seems reasonable. However, the implications on the arithmetic are substantial as we can see during the second step of Gaussian elimination, where we replace E_3 by

$$\begin{aligned} &\overbrace{[0,(2-\epsilon^{-1}),(1-\epsilon^{-1}),(4-\epsilon^{-1})]}^{E_3} - \frac{(2-\epsilon^{-1})}{(1-\epsilon^{-1})} \overbrace{[0,(1-\epsilon^{-1}),(1-\epsilon^{-1}),(1-\epsilon^{-1})]}^{E_2} \\ &\qquad = \Big[0,\, 0,\, (1-\epsilon^{-1}) - (2-\epsilon^{-1}), (4-\epsilon^{-1}) - (2-\epsilon^{-1})\Big] \end{aligned} \tag{2.18}$$

Notice that if we use exact arithmetic we have

$$(1-\epsilon^{-1}) - (2-\epsilon^{-1}) = -1$$
$$(4-\epsilon^{-1}) - (2-\epsilon^{-1}) = 2$$

which enable us to reduce the computations above to $E_3 = [0,0,-1,2]$. However, for the floating point system:

```python
print( f"{(1-1/eps)-(2 - 1/eps):.55f}")
print( f"{(4-1/eps)-(2 - 1/eps):.55f}")
```

```
-2.0000000000000000000000000000000000000000000000000000000
2.0000000000000000000000000000000000000000000000000000000
```

and thus in floating point arithmetic that is $E_3 = [0.0, 0.0, -2.0, 2.0]$. When we use this (very bad) approximation during back substitution we obtain the inaccurate solution from above.

The numerical issue that we encountered above is called *catastrophic cancellation* that can occur when we add two real numbers of very similar size. Note that given $x, y \in \mathbb{R}$ with $x \neq 0$ and $y \neq 0$ we may define

$$\epsilon_x = \frac{\mathrm{fl}(x) - x}{x}, \qquad \epsilon_y = \frac{\mathrm{fl}(y) - y}{y} \tag{2.19}$$

and from Equation 2.17 ww know that $|\epsilon_x| \leq \epsilon_{mach}$ and $|\epsilon_y| \leq \epsilon_{mach}$. Similarly, define

$$w = \mathrm{fl}(y) - \mathrm{fl}(x) \tag{2.20}$$

and

$$\epsilon_w = \frac{\mathrm{fl}(w) - w}{w} \tag{2.21}$$

with $|\epsilon_w| \leq \epsilon_{mach}$. In Exercise 2.9 we show that the relative error of the floating point of $y - x$ is

$$\frac{\mathrm{fl}(\mathrm{fl}(y) - \mathrm{fl}(x)) - (y - x)}{y - x} = \left[\frac{y\epsilon_y - x\epsilon_x}{y - x}\right](\epsilon_w + 1) + \epsilon_w. \tag{2.22}$$

Therefore, unless both x and y are represented exactly as floats (i.e. $\epsilon_x = \epsilon_y$), the relative error of the subtraction can be substantially large if $(y - x)$ is very small relative to the values of y and x.

We now look at techniques that we apply during forward elimination in order to avoid loss of accuracy due to finite point arithmetic. We implement these techniques at the start of step i prior to the usual row operators.

2.2.4 Partial pivoting

Partial pivoting is aimed at avoiding dividing by a small pivot. More specifically, at the ith step before making zeros below the pivot element a_{ii} we find the smallest p $(i \leq p \leq n)$ such that

$$|a_{pi}| = \max_{i \leq j \leq n} |a_{ji}|,$$

and then we swap rows i and p, i.e. $E_p \leftrightarrow E_i$. The pseudocode is given in Algorithm 2.4 .

The complexity of Gaussian elimination with partial pivoting is also $\mathcal{O}(n^3)$. Note that the number of elementary arithmetic operations is the same as Algorithm 2.3 . The number of comparison required for partial pivoting is computed in Exercise 2.14.

Exercise 2.7. Applied Gaussian elimination with partial pivoting to solve the linear system of Equation 2.16.

Solution

Let $i = 1$. We need to compute the smallest p $(1 \leq p \leq n)$ such that

$$|a_{p1}| = \max_{1 \leq j \leq n} |a_{j1}| = \max\{\overbrace{|a_{11}|}^{\epsilon}, \overbrace{|a_{21}|}^{1}, \overbrace{|a_{31}|}^{1}\} = \max\{\epsilon, 1, 1\}$$

Note that both $|a_{21}|$ and $|a_{31}|$ are the maximum above but we take the smallest integer p. Hence $p = 2$ in this case. Therefore, partial pivoting indicates that, before applying Gaussian elimnation, we need to swap rows E_2 and E_1:

$$\tilde{A} \sim \begin{bmatrix} 1 & 1 & 1 & 1 \\ \epsilon & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix}$$

The first step of Gaussian elimination (please verify steps):

$$\tilde{A} \sim \begin{bmatrix} 1 & 1 & 1 & 1 \\ \epsilon & 1 & 1 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1-\epsilon & 1-\epsilon & 1-\epsilon \\ 0 & 1 & 0 & 3 \end{bmatrix}$$

Before we proceed to the second step, we conduct partial pivoting, i.e find the smallest p such that (recall now $i = 2$)

$$|a_{p2}| = \max_{2 \leq j \leq n} |a_{j2}| = \max\{\overbrace{|a_{22}|}^{1-\epsilon}, \overbrace{|a_{32}|}^{1}\} = \max\{1 - \epsilon, 1\}$$

since $\epsilon \ll 1$ then the maximum is attained for $p = 3$ hence we swap E_2 and E_3, i.e.

$$\tilde{A} \sim \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1-\epsilon & 1-\epsilon & 1-\epsilon \\ 0 & 1 & 0 & 3 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 1-\epsilon & 1-\epsilon & 1-\epsilon \end{bmatrix}$$

and proceed to the second step of Gaussian elimination

$$\tilde{A} \sim \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & (1-\epsilon) & (1-\epsilon) & (1-\epsilon) \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & (1-\epsilon)-(1-\epsilon) & -\frac{(1-\epsilon)}{(1-\epsilon)} & 3-\frac{(1-\epsilon)}{(1-\epsilon)} \end{bmatrix}$$
$$\sim \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

You can easily verify that the solution of the system is Equation 2.15. Although we have used exact arithmetic for these calculations, when this procedure is computed in floating point arithmetic, no catastrophic cancellation occurs. While the computer uses a floating point representation of $(1 - \epsilon)$ the subsequnt operations are stable since these involve floating point numbers. Note for example that

```
x=(1-eps)
y=3-x/x
print(f"{y:.55f}")
```

```
2.0000000000000000000000000000000000000000000000000000000
```

In Listing 2.8 we implement and test the partial pivoting strategy from Algorithm 2.4 . Note that we obtain an accurate solution to Equation 2.16 with $\epsilon = 10^{-16}$.

! Important 17: Computing Activity

To find the row with largest pivot, in lines 30-36 of Listing 2.8 we used a `for` loop over the rows below the original pivot. This can be done more efficiently using the built-in `numpy` function `argmax` which gives the index of the maximal element of an array (along one particular `axis`). The following example shows what this function does on a 1D array:

```
import numpy as np

array1D=np.array([1,4,10,23,2,1,6,8,10],dtype=float)

amax=np.argmax(array1D)

print(f"index of maximal element: {amax}")
```

```
index of maximal element: 3
```

For a 2D array, the following code finds the maximum over the second column:

```
array2D=np.array([[2,3,4],[4,9,1],[60,15,2]], dtype=float)
print(f"the 2D array is:\n {array2D}")
amax=np.argmax(array2D[:,1])

print(f"index of maximal element in second column:{amax}")
```

```
the 2D array is:
 [[ 2.  3.  4.]
 [ 4.  9.  1.]
 [60. 15.  2.]]
index of maximal element in second column:2
```

and for the third column:

```
amax=np.argmax(array2D[:,2])

print(f"index of maximal element in third column:{amax}")
```

```
index of maximal element in third column:0
```

you can also use this to find the maximum over a row (say the firs one)

```
amax=np.argmax(array2D[0,:])

print(f"index of maximal element in first row:{amax}")
```

```
index of maximal element in first row:2
```

You can read more about `np.argmax` here.
Modify the code from Listing 2.8 to use `np.argmax` (in a suitable way) to avoid using the `for` loop for the computation of the pivot. Test your code to make sure it provides the same output.

2.2.5 Scaled partial pivoting {#sec-partial-pivoting}.

There are instances where partial pivoting is not enough to avoid loss of accuracy. This is particularly the case when the system that we want to solve is poorly scaled. For example, consider the system

$$A = \begin{bmatrix} 1 & \epsilon^{-1} & \epsilon^{-1} \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} \epsilon^{-1} \\ 1 \\ 2 \end{bmatrix} \tag{2.23}$$

for small $\epsilon > 0$ say $\epsilon = 10^{-16}$ as before. The augmented matrix is

$$\tilde{A} = \left[\begin{array}{ccc|c} 1 & \epsilon^{-1} & \epsilon^{-1} & \epsilon^{-1} \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{array}\right]$$

Let us apply Algorithm 2.4 to reduce this matrix. Let $i = 1$ and compute the smallest p $(1 \le p \le n)$ such that

$$|a_{p1}| = \max_{1\le j\le n} |a_{j1}| = \max\{\overbrace{|a_{11}|}^{1}, \overbrace{|a_{21}|}^{1}, \overbrace{|a_{31}|}^{1}\} = \max\{1, 1, 1\}$$

Since the smallest integer is $p = 1$, we do not need to swap any of the rows. Hence

$$\tilde{A} = \begin{bmatrix} 1 & \epsilon^{-1} & \epsilon^{-1} & \epsilon^{-1} \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \sim \begin{bmatrix} 1 & \epsilon^{-1} & \epsilon^{-1} & \epsilon^{-1} \\ 0 & 1-\epsilon^{-1} & 2-\epsilon^{-1} & 1-\epsilon^{-1} \\ 0 & 1-\epsilon^{-1} & 1-\epsilon^{-1} & 2-\epsilon^{-1} \end{bmatrix}$$

For $i = 2$ we have

$$|a_{p2}| = \max_{2\le j\le n} |a_{j2}| = \max\{\overbrace{|a_{22}|}^{\epsilon^{-1}-1}, \overbrace{|a_{32}|}^{\epsilon^{-1}-1}\} = \max\{\epsilon^{-1}-1, \epsilon^{-1}-1\} = \epsilon^{-1}-1$$

Therefore no row swapping is needed. We now conduct Gaussian elimnation:

$$\begin{aligned} \tilde{A} &\sim \begin{bmatrix} 1 & \epsilon^{-1} & \epsilon^{-1} & \epsilon^{-1} \\ 0 & 1-\epsilon^{-1} & 2-\epsilon^{-1} & 1-\epsilon^{-1} \\ 0 & 1-\epsilon^{-1} & 1-\epsilon^{-1} & 2-\epsilon^{-1} \end{bmatrix} \\ &\sim \begin{bmatrix} 1 & \epsilon^{-1} & \epsilon^{-1} & \epsilon^{-1} \\ 0 & 1-\epsilon^{-1} & 2-\epsilon^{-1} & 1-\epsilon^{-1} \\ 0 & 0 & (1-\epsilon^{-1})-(2-\epsilon^{-1}) & (2-\epsilon^{-1})-(1-\epsilon^{-1}) \end{bmatrix}, \end{aligned} \tag{2.24}$$

which with exact arithmetic we can be further reduced to

$$\tilde{A} \sim \begin{bmatrix} 1 & \epsilon^{-1} & \epsilon^{-1} & \epsilon^{-1} \\ 0 & 1-\epsilon^{-1} & 2-\epsilon^{-1} & 1-\epsilon^{-1} \\ 0 & 0 & -1 & 1 \end{bmatrix}. \tag{2.25}$$

In Exercise 2.10 you will show that

$$\mathbf{x} = \begin{bmatrix} \frac{\epsilon^{-1}}{\epsilon^{-1}-1} \\ \frac{2\epsilon^{-1}-3}{\epsilon^{-1}-1} \\ -1 \end{bmatrix}, \tag{2.26}$$

is the solution, and that for sufficiently large ϵ becomes

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}, \tag{2.27}$$

When using floating point arithmetic, the representation of Equation 2.24 does not lead to the one for Equation 2.25 since catastrophic cancellation occurs. We can verify this via:

```
A=np.array([[1,1e16,1e16],[1,1,2],[1,1,1]])
b=np.array([[1e16],[1],[2]])
tildeA=   gaussian_elimination_partial_pivoting(A, b)
x=backward_substitution(tildeA)
print(tildeA)
print(x)
```

```
[[ 1.e+00  1.e+16  1.e+16  1.e+16]
 [ 0.e+00 -1.e+16 -1.e+16 -1.e+16]
 [ 0.e+00  0.e+00 -2.e+00  2.e+00]]
[[ 0.]
 [ 2.]
 [-1.]]
```

To address the above issue we introduce **scaled partial pivoting** where the goal is to swap rows so that the pivot has the largest value relative to the other values of the same row. To this end, we first define scale factors

$$s_i = \max_{1\le j\le n} |a_{ij},| \qquad i = 1, \dots, n.$$

These scale factors are computed at the beginning of the algorithm. Then, at the *ith* step we find smallest p $(i \le p \le n)$ such that

$$\frac{|a_{pi}|}{s_p} = \max_{i\le j\le n} \frac{|a_{ji}|}{s_j},$$

and perform $E_p \leftrightarrow E_i$ and $s_p \leftrightarrow s_i$.

The pseudocode for scaled pivoting is presented in Algorithm 2.5 .

The complexity of Gaussian elimination with scaled pivoting is also $\mathcal{O}(n^3)$.The numbers of comparison and additional divisons is $\mathcal{O}(n^2)$. This is computed in Exercise 2.15.

Exercise 2.8. Apply scaled pivoting to Equation 2.23 and show that the solution of the linear system is given by Equation 2.26. Explain why catastrophic cancellation does not occur in this case.

Solution

Scale factors are

$$s_1 = \epsilon^{-1}, \qquad s_2 = 2, \qquad s_3 = 1$$

Let $i = 1$ and note that

$$\frac{|a_{p1}|}{s_p} = \max_{1\le j\le n} \frac{|a_{j1}|}{s_j} = \max\left\{\frac{|a_{11}|}{s_1}, \frac{|a_{21}|}{s_2}, \frac{|a_{31}|}{s_3}\right\} = \max\{\epsilon, 1/2, 1\}$$

hence $p = 3$, and so we conduct $E_3 \leftrightarrow E_1$ and $s_3 \leftrightarrow s_1$. We now conduct Gaussian elimination:

$$\tilde{A} = \begin{bmatrix} 1 & \epsilon^{-1} & \epsilon^{-1} & \epsilon^{-1} \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 & 2 \\ 1 & 1 & 2 & 1 \\ 1 & \epsilon^{-1} & \epsilon^{-1} & \epsilon^{-1} \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \\ 0 & \epsilon^{-1}-1 & \epsilon^{-1}-1 & \epsilon^{-1}-2 \end{bmatrix}$$

We now compute

$$\frac{|a_{p2}|}{s_p} = \max_{2\le j\le n} \frac{|a_{j2}|}{s_j} = \max\left\{\frac{|a_{22}|}{s_2}, \frac{|a_{32}|}{s_3}\right\} = \max\left\{0, \frac{|\epsilon^{-1}-1|}{\epsilon^{-1}}\right\}$$

and we thus perform $E_2 \leftrightarrow E_3$ and $s_2 \leftrightarrow s_3$. to obtain

$$\tilde{A} =\sim \begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \\ 0 & \epsilon^{-1}-1 & \epsilon^{-1}-1 & \epsilon^{-1}-2 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & \epsilon^{-1}-1 & \epsilon^{-1}-1 & \epsilon^{-1}-2 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

which is already in reduced form. We can apply back substitution to find $x_3 = -1$, then

$$(\epsilon^{-1}-1)x_2 = (\epsilon^{-1}-2) - (\epsilon^{-1}-1)x_3 = (\epsilon^{-1}-2) + (\epsilon^{-1}-1)$$

and so

$$x_2 = \frac{(\epsilon^{-1}-2)+(\epsilon^{-1}-1)}{(\epsilon^{-1}-1)} = \frac{2\epsilon^{-1}-3}{\epsilon^{-1}-1}$$

and

$$x_1 = 2 - x_3 - x_2 = 2 + 1 - \frac{(\epsilon^{-1}-2)+(\epsilon^{-1}-1)}{(\epsilon^{-1}-1)} = \frac{\epsilon^{-1}}{\epsilon-1}$$

which is the solution from Equation 2.26. Note that while we have used exact arithmetic to arrive at this solution, we can see that no catastrophic cancellation has occurred if these computations are performed via floating point arithmetic.

Using a python implementation (**not provided here**) of scaled pivoting via a function `gaussian_elimination_scaled_` `b)` we can test that for $\epsilon = 10^{-16}$, the solution is correct:

```
A=np.array([[1,1e16,1e16],[1,1,2],[1,1,1]])
b=np.array([[1e16],[1],[2]])
tildeA=   gaussian_elimination_scaled_pivoting(A, b)
x=backward_substitution(tildeA)
x1, x2, x3=x[0],x[1], x[2]
print(tildeA)
print(x)
#print( f"{x1:.55f}")
#print( f"{x2:.55f}")
#print( f"{x3:.55f}")
```

```
swaping row  1  with row  3
swaping row  2  with row  3
[[ 1.e+00  1.e+00  1.e+00  2.e+00]
 [ 0.e+00  1.e+16  1.e+16  1.e+16]
 [ 0.e+00  0.e+00  1.e+00 -1.e+00]]
[[ 1.]
 [ 2.]
 [-1.]]
```

2.2.6 Additional Problems

Exercise 2.9. Derive Equation 2.22.

Solution

From Equation 2.19 we have that

$$\text{fl}(x) = x(\epsilon_x + 1), \qquad \text{fl}(y) = y(\epsilon_y + 1)$$

Then, we can write Equation 2.20 as

$$w = \text{fl}(y) - \text{fl}(x) = y(\epsilon_y + 1) - x(\epsilon_x + 1) = y\epsilon_y - x\epsilon_x + (y - x) \tag{2.28}$$

We rewrite Equation 2.21 as

$$\mathrm{fl}(w) = w(\epsilon_w + 1)$$

and substituting Equation 2.28 in the previous expression yields:

$$\mathrm{fl}(\mathrm{fl}(y) - \mathrm{fl}(x)) = \Big[y\epsilon_y - x\epsilon_x + (y-x)\Big](\epsilon_w + 1)$$

that we can write as

$$\mathrm{fl}(\mathrm{fl}(y) - \mathrm{fl}(x)) - (y-x) = \Big[y\epsilon_y - x\epsilon_x + (y-x)\Big](\epsilon_w + 1) - (y-x)$$

and thus

$$\frac{\mathrm{fl}(\mathrm{fl}(y) - \mathrm{fl}(x)) - (y-x)}{y-x} = \left[\frac{y\epsilon_y - x\epsilon_x}{y-x} + 1\right](\epsilon_w + 1) - 1$$

This can be rewritten a Equation 2.22.

Exercise 2.10. Apply backward substitution to Equation 2.25 to find that the solution of the linear system is the one give in Equation 2.26 and that the limit as $\epsilon \to 0$ is Equation 2.27. Use exact arithmetic.

Solution

Back substitution to Equation 2.25:

$$\begin{aligned}
x_3 &= -1\\
x_2 &= \frac{1}{1-\epsilon^{-1}}\Big[(1-\epsilon^{-1}) - (2-\epsilon^{-1})(-1)\Big] = 1 + \frac{(2-\epsilon^{-1})}{1-\epsilon^{-1}} = \frac{(3-2\epsilon^{-1})}{1-\epsilon^{-1}}\\
x_1 &= \epsilon^{-1} - \epsilon^{-1}x_3 - \epsilon^{-1}x_2 = \epsilon^{-1}\left[2 - \frac{(3-2\epsilon^{-1})}{1-\epsilon^{-1}}\right] = \frac{-\epsilon^{-1}}{1-\epsilon^{-1}}
\end{aligned}$$

Note that

$$x_2 = \frac{(3\epsilon - 2)}{\epsilon - 1} \to 2 \qquad \text{as} \quad \epsilon \to 0$$

and

$$x_1 = \frac{-1}{\epsilon - 1} \to 1 \qquad \text{as} \quad \epsilon \to 0$$

Hence Equation 2.26 converges to Equation 2.27 as $\epsilon \to 0$.

Exercise 2.11. Consider the following linear system:

$$\begin{aligned}
x_1 - 5x_2 + x_3 &= \quad 7\\
10x_1 + 20x_3 &= \quad 6\\
5x_1 - x_3 &= \quad 4
\end{aligned}$$

1. Find the row interchanges that are required to solve each system using Algorithm 2.3 (i.e. standard pivoting).
2. Find the row interchanges that are required to solve each system using Algorithm 2.4 (partial pivoting).
3. Find the row interchanges that are required to solve each system using Algorithm 2.5 (partial pivoting) .

You can assume exact arithmetic.

Solution

1. Gaussian elimination with standard pivoting. Firs step: pivot $a_{11} = 1 \neq 0$ so we do: $E_2 \to E_2 - (10/1)E_1$ and $E_3 \to E_3 - (5/1)E_1$:

$$\tilde{A} = \begin{bmatrix} 1 & -5 & 1 & 7\\ 10 & 0 & 20 & 6\\ 5 & 0 & -1 & 4 \end{bmatrix} \sim \begin{bmatrix} 1 & -5 & 1 & 7\\ 0 & 50 & 10 & -64\\ 0 & 25 & -6 & -31 \end{bmatrix}$$

Second step: pivot $a_{22} = 50 \neq 0$ so we do $E_3 \to E_3 - (25/50)E_2$

$$\tilde{A} \sim \begin{bmatrix} 1 & -5 & 1 & 7 \\ 0 & 50 & 10 & -64 \\ 0 & 25 & -6 & -31 \end{bmatrix} \sim \begin{bmatrix} 1 & -5 & 1 & 7 \\ 0 & 50 & 10 & -64 \\ 0 & 0 & -11 & 1 \end{bmatrix}$$

Check with code:

```
A=np.array([[1,-5,1],[10,0,20],[5,0,-1]], dtype=float)
b=np.array([[7],[6],[4]], dtype=float)
tildeA=   gaussian_elimination(A, b)
print(tildeA)
```

```
[[   1.  -5.    1.    7.]
 [   0.  50.   10. -64.]
 [   0.    0. -11.    1.]]
```

2. Gaussian elimination with partial pivoting. Find:

$$|a_{p1}| = \max_{1\le j\le n} |a_{j1}| = \max\{\overbrace{|a_{11}|}^{1}, \overbrace{|a_{21}|}^{10}, \overbrace{|a_{31}|}^{5}\} = \max\{1, 10, 5\}$$

hence $p = 2$ and so $E_1 \leftrightarrow E_2$:

$$\tilde{A} = \begin{bmatrix} 1 & -5 & 1 & 7 \\ 10 & 0 & 20 & 6 \\ 5 & 0 & -1 & 4 \end{bmatrix} \sim \begin{bmatrix} 10 & 0 & 20 & 6 \\ 1 & -5 & 1 & 7 \\ 5 & 0 & -1 & 4 \end{bmatrix}$$

Then: $E_2 \to E_2 - (1/10)E_1$ and $E_3 \to E_3 - (5/10)E_1$:

$$\tilde{A} \sim \begin{bmatrix} 10 & 0 & 20 & 6 \\ 1 & -5 & 1 & 7 \\ 5 & 0 & -1 & 4 \end{bmatrix} \sim \begin{bmatrix} 10 & 0 & 20 & 6 \\ 0 & -5 & 1 & (32/5) \\ 0 & 0 & -11 & 1 \end{bmatrix}$$

Check with code:

```
A=np.array([[1,-5,1],[10,0,20],[5,0,-1]], dtype=float)
b=np.array([[7],[6],[4]], dtype=float)
tildeA=   gaussian_elimination_partial_pivoting(A, b)
print(tildeA)
```

```
swaping row  1  with row  2
[[ 10.    0.   20.    6. ]
 [  0.   -5.   -1.    6.4]
 [  0.    0.  -11.    1. ]]
```

3. Gaussian elimination with scaled pivoting. Compute scale factors (maximum of absolute valute over each column)

$$s_1 = 5, \qquad s_2 = 20, \qquad s_3 = 5$$

First step. We find

$$\frac{|a_{p1}|}{s_p} = \max_{1\le j\le n} \frac{|a_{j1}|}{s_j} = \max\left\{\frac{|a_{11}|}{s_1}, \frac{|a_{21}|}{s_2}, \frac{|a_{31}|}{s_3}\right\} = \max\{1/5, 10/20, 5/5\}$$

so $p = 3$ so $E_1 \leftrightarrow E_3$ and so $s_1 \leftrightarrow s_3$:

$$\tilde{A} = \begin{bmatrix} 1 & -5 & 1 & 7 \\ 10 & 0 & 20 & 6 \\ 5 & 0 & -1 & 4 \end{bmatrix} \sim \begin{bmatrix} 5 & 0 & -1 & 4 \\ 10 & 0 & 20 & 6 \\ 1 & -5 & 1 & 7 \end{bmatrix}$$

and $s_1 = 5, \qquad s_2 = 20, \qquad s_3 = 5$. Now we do $E_2 \to E_2 - (10/5)E_1$ and $E_3 \to E_3 - (1/5)E_1$ so

$$\tilde{A} \sim \begin{bmatrix} 5 & 0 & -1 & 4 \\ 10 & 0 & 20 & 6 \\ 1 & -5 & 1 & 7 \end{bmatrix} \sim \begin{bmatrix} 5 & 0 & -1 & 4 \\ 0 & 0 & 22 & -2 \\ 0 & -5 & 6/5 & 31/5 \end{bmatrix}$$

$$\frac{|a_{p2}|}{s_p} = \max_{2\le j\le n} \frac{|a_{j2}|}{s_j} = \max\left\{\frac{|a_{22}|}{s_2}, \frac{|a_{32}|}{s_3}\right\} = \max\{0/20, 5/5\}$$

hence $p = 3$ so $E_2 \leftrightarrow E_3$ (we do not need to swap scaling factors as we won't need them):

$$\tilde{A} \sim \begin{bmatrix} 5 & 0 & -1 & 4 \\ 0 & 0 & 22 & -2 \\ 0 & -5 & 6/5 & 31/5 \end{bmatrix} \sim \begin{bmatrix} 5 & 0 & -1 & 4 \\ 0 & -5 & 6/5 & 31/5 \\ 0 & 0 & 22 & -2 \end{bmatrix}$$

check with code:

```
A=np.array([[1,-5,1],[10,0,20],[5,0,-1]], dtype=float)
b=np.array([[7],[6],[4]], dtype=float)
tildeA=   gaussian_elimination_scaled_pivoting(A, b)
print(tildeA)
```

```
swaping row  1  with row  3
swaping row  2  with row  3
[[ 5.   0.  -1.   4. ]
 [ 0.  -5.   1.2  6.2]
 [ 0.   0.  22.  -2. ]]
```

Exercise 2.12. Consider the following linear system:

$$\begin{aligned} x_1 + x_2 - 3x_3 &= \quad 1 \\ x_1 + x_2 + 4x_3 &= \quad 2 \\ 2x_1 - x_2 + 2x_3 &= \quad 3 \end{aligned}$$

- Find the row interchanges that are required to solve each system using Algorithm 2.3 (i.e. standard pivoting). You can assume exact arithmetic.
- Find the row interchanges that are required to solve each system using Algorithm 2.4 (partial pivoting) . You can assume exact arithmetic.
- Find the row interchanges that are required to solve each system using Algorithm 2.5 (partial pivoting) . You can assume exact arithmetic.

Solution

```
A=np.array([[1,-5,1],[10,0,20],[5,0,-1]], dtype=float)
b=np.array([[7],[6],[4]], dtype=float)
tildeA=   gaussian_elimination_scaled_pivoting(A, b)
x=backward_substitution(tildeA)
print(x)
#print( f"{x1:.55f}")
#print( f"{x2:.55f}")
#print( f"{x3:.55f}")
```

```
swaping row  1  with row  3
swaping row  2  with row  3
[[ 0.78181818]
 [-1.26181818]
 [-0.09090909]]
```

Exercise 2.13. To find $c_p = \max\{c_1, c_2, \dots, c_m\}$, show that the number of comparisons between two numbers equals~$m - 1$.

Solution

A straightforward way to compute $c_p = \max\{c_1, c_2, \dots, c_m\}$ is via a for loop: we start with $c_p = c_1$

```
for $i = 2$ to $m$
    if $c_p > c_i$
$c_p = c_i$
    end if
end for
```

so we require 1 comparison (if $c_p > c_i$) for $i = 2 \to m$, i.e. $m - 1$ times.

Exercise 2.14. Compute the total number of comparisons carried out using partial pivoting.

Solution

A straightforward way to compute the smallest p such that $|a_{pi}| = \max_{i \le j \le n} |a_{ji}|$ is via a for loop: We start with $p = i$ and $|a_{pi}| = |a_{ii}|$

```
for   $k = i + 1$ to $n$
      if $|a_{ki}| > |a_{pi}|$
          $p = i + 1$
      end if
end for
```

this requires $n - (i + 1) + 1 = (n - i)$ comparisons. Since $i = 1 \to n - 1$ we have

$$\begin{aligned}\mathcal{C}_{com}(n) &= \sum_{i=1}^{n-1} (n - i) \\ &= \sum_{i=1}^{n-1} m \\ &= \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}\end{aligned}$$

Exercise 2.15. Compute the total number of comparisons and divisions carried out using scaled pivoting (only for the pivoting strategy).

Solution

We first note that in line 7 of Algorithm 2.5 we compute the scaling factor $s_i = \max_{1 \le j \le n} |a_{ij},|$ which from Problem **??** requires $n - 1$ comparisons. Since we have $(i = 1, \dots, n)$ then the complexity of scaling factors is

$$\sum_{i=1}^{n} (n-1) = (n-1) \sum_{i=1}^{n} 1 = (n-1)n = n^2 - n$$

In line 13 of Algorithm 2.5 we need to compute smallest p such that $\frac{|a_{pi}|}{s_p} = \max_{i \le j \le n} \frac{|a_{ji}|}{s_j}$. We start with $p = i$ and $\xi = \frac{|a_{ii}|}{s_i}$ (1 division)

```
for   $k = i + 1$ to $n$
      $\beta = \frac{|a_{ki}|}{s_k}$  (1 division)
      if $\beta > \xi$  (1 comparison)
         $p = i + 1$
         $\xi = \beta$
      end if
end for
```

This requires $n-(i+1)+1=(n-i)$ comparisons and $(n-i)$ divisions. Since $i = 1 \to n-1$ we have

$$\begin{aligned}\mathcal{C}_{com}(n) &= \sum_{i=1}^{n-1}(n-i)\\ &= \sum_{i=1}^{n-1} m\\ &= \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}\end{aligned}$$

and also

$$\begin{aligned}\mathcal{C}_{div,piv}(n) &= \sum_{i=1}^{n-1}(n-i+1)\\ &= \sum_{i=1}^{n-1}(n-i) + \sum_{i=1}^{n-1} 1 = \sum_{i=1}^{n-1} m + (n-1)\\ &= \frac{(n-1)n}{2} + (n-1) = \frac{n^2}{2} - \frac{n}{2} + n - 1 = \frac{n^2}{2} + \frac{n}{2} - 1\end{aligned}$$

So total number of comparisons (including those for computing scale factors):

$$\mathcal{C}_{com}(n) = n^2 - n + \frac{n^2}{2} - \frac{n}{2} = \frac{3}{2}n(n-1)$$

The total number of divisions is

$$\mathcal{C}_{div,scaled}(n) = \frac{n^2}{2} - \frac{n}{2}$$

2.3 Matrix Factorisation

Suppose that we want to solve a collection of M linear systems of the form

$$A\mathbf{x}^m = \mathbf{b}^m, \qquad m = 1, \dots, M \tag{2.29}$$

where we note that matrix A is the same for each $m = 1, \dots, M$. Such a situation arises when solving linear system obtained from discretising differential equations. If A is such that Gaussian elimination with backward substitution can be applied to solve each system. Since the computational cost of Gaussian elimination with back substitution is $\mathcal{O}(n^3)$, the cost of solving Equation 2.29 is $\mathcal{O}(m\,n^3)$.

Note that if we were able to compute the inverse A^{-1} of A, The systems in Equation 2.29 can be solved via

$$\mathbf{x}^m = A^{-1}\mathbf{b}^m, \qquad m = 1, \dots, M \tag{2.30}$$

which requires m matrix-vector multiplications. Because the complexity of a matrix-vector multiplication is $\mathcal{O}(n^2)$ the complexity of Equation 2.30 is $\mathcal{O}(mn^2)$ and hence smaller than employing Gaussian elimination. Unfortunately, computing the inverse A^{-1} can be very costly. Cramer's method, for example, requires $(n+1)n!$ operations which is unfeasible for large n.

In this section we introduce factorisation methods that will enable us to express A in the following form

$$A = LU \tag{2.31}$$

where L a **lower triangular** matrix and U an **upper-triangular** matrix as shown in Figure 2.9 and more properly defined in Definition 2.1.

Definition 2.1. A matrix U is triangular if all its non-zero elements are either on the main diagonal or to one side of it. There are two possibilities:

- A matrix $U = [u_{ij}]$ is upper triangular if $u_{ij} = 0$ for all $j < i$
- A matrix $L = [l_{ij}]$ is lower triangular if $l_{ij} = 0$ for all $j > i$

An example of an upper triangular matrix is the matrix that we obtained via Gaussian elimination Algorithm 2.3
.

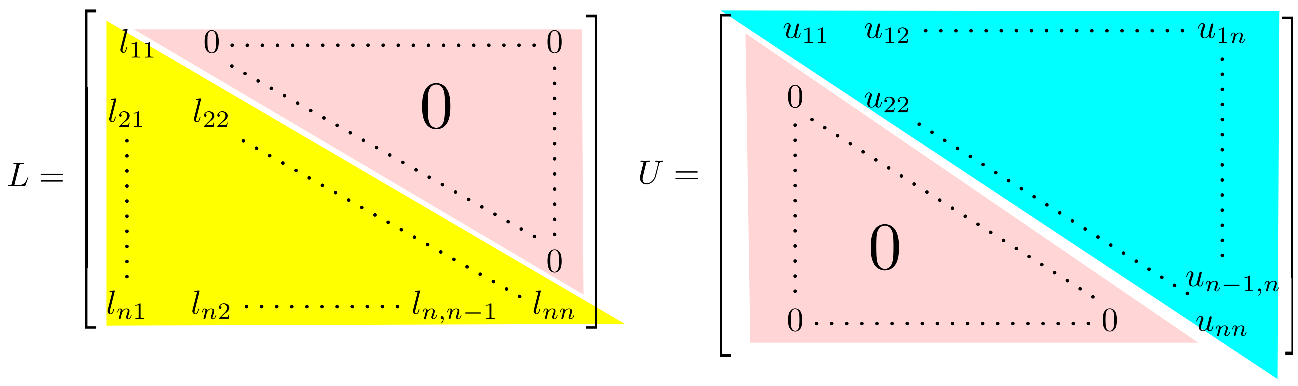


Figure 2.9: L and U matrices

As we will see below, having the LU factorisation is similar to having the inverse of A, in the sense that, once the LU factorisation is obtained, the system from Equation 2.30 can be computed with $\mathcal{O}(mn^2)$ operations.

The representation of A given in Equation 2.31 is called LU-factorisation. While not every matrix A admits such a factorisation, many matrices that arise in numerical methods can be represented via Equation 2.31.

Note

It is worth mentioning that, in general, an LU factorisation is not unique. If a matrix A can be express via $A = LU$, it can also be expressed, for any diagonal matrix D as

$$A = LU = L\,\overbrace{(DD^{-1})}^{I}\,U = (LD)(LD^{-1}).$$

However, there are specific forms of L and U

Before we dive into the details of the algorithm. Let us start with some motivating example

Exercise 2.16.

1. Determine the LU factorization for
$$A = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$
where $L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}$ and $U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$.

2. What happens for $B = \begin{bmatrix} 0 & -1 & -1 \\ 2 & 1 & -1 \\ 1 & 1 & 0 \end{bmatrix}$?

Solution

1. We need
$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & u_{12}l_{21} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

Equating first rows gives:
$$u_{11} = 1, \qquad u_{12} = 1, \qquad u_{13} = 0$$

Equating second rows:
$$l_{21}u_{11} = 2, \qquad u_{12}l_{21} + u_{22} = 1, \quad l_{21}u_{13} + u_{23} = -1$$

from which we obtained
$$l_{21} = 2, \qquad u_{22} = 1 - 2 = -1, \quad u_{23} = -1$$

Equating third rows:

$$l_{31}u_{11} = 0, \qquad l_{31}u_{12} + l_{32}u_{22} = -1, \qquad l_{31}u_{13} + l_{32}u_{23} + u_{33} = -1$$

and so we solve for:

$$l_{31} = 0, \qquad l_{32} = 1, \qquad u_{33} = -1 - l_{32}u_{23} = 0$$

Therefore

$$\underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 0 & -1 & -1 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 0 \end{bmatrix}}_{U}$$

2. Note that $A \sim B$ since we can swap: $E_1 \leftrightarrow E_3$. However, we cannot write $B = LU$ since if we try, we find that

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & u_{12}l_{21} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{bmatrix} = \begin{bmatrix} 0 & -1 & -1 \\ 2 & 1 & -1 \\ 1 & 1 & 0 \end{bmatrix}$$

so $0 = u_{11}$ and $2 = l_{21}u_{11} = 0$ which is a contradiction.

2.3.1 Forward and Backward substitution

Knowing the factorisation $A = LU$ is like having the inverse matrix A^{-1}. Once we have $A = LU$ we can solve the linear system

$$LU\mathbf{x} = \mathbf{b} \tag{2.32}$$

for any $b \in \mathbb{R}^n$ (if a solution exits).

To solve the system $A\mathbf{x} = \mathbf{b}$ using the LU factorisation we first define the new variable

$$\mathbf{y} = U\mathbf{x} \tag{2.33}$$

and note that the system from Equation 2.32 can be written as

$$L\mathbf{y} = \mathbf{b} \tag{2.34}$$

To solve the system from Equation 2.32 we conduct the two steps:

- Step 1: Solve Equation 2.34 to find $\mathbf{y}$
- Step 2: Solve Equation 2.33 to find $\mathbf{x}$.

Note that for a $n \times n$ system, Equation 2.34 can be written as

$$\begin{aligned} l_{11}y_1 &= b_1 \\ l_{21}y_1 + l_{22}y_2 &= b_2 \\ \vdots &= \vdots \\ l_{i1}y_1 + l_{i2}y_2 \cdots + l_{ii}y_i &= b_i \\ \vdots &= \vdots \\ l_{n1}y_1 + l_{n2}y_2 + \cdots + l_{n,n-1}y_{n-1} + l_{nn}y_n &= b_n \end{aligned}$$

which can be easily solved via **forward substitution**:

$$\begin{aligned} y_1 &= b_1/l_{11} \\ y_2 &= (b_2 - l_{21}y_1)/l_{22} \\ \vdots &= \vdots \\ y_i &= (b_i - \sum_{j=1}^{i-1} l_{ij}y_j)/l_{ii} \\ \vdots &= \vdots \\ y_n &= (b_n - \sum_{j=1}^{n-1} l_{nj}y_j)/l_{nn} \end{aligned}$$

Once this system is solved for $\mathbf{y}$ we can solve Equation 2.33 for $\mathbf{x}$:

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + \cdots + \cdots + u_{1n}x_n &= y_1 \\ u_{22}x_2 + u_{23}x_3 + \cdots + \cdots + u_{2n}x_n &= y_2 \\ \vdots &= \vdots \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}u_n &= y_{n-1} \\ u_{nn}x_n &= y_n \end{aligned}$$

using the backward substitution that we introduce in Algorithm 2.2 .

We know that the complexity of backward substitution is $\mathcal{O}(n^2)$. It is not difficult to see that forward substitution involves the same number of arithmetic operations and hence its complexity is also $\mathcal{O}(n^2)$. Hence, as discussed earlier, once we have computed the LU factorisation of A, the cost of solving the system $A\mathbf{x} = \mathbf{b}$ is $\mathcal{O}(n^2)$.

Note

A $n \times n$ matrix A can admit LU factorisation even though the linear system $A\mathbf{x} = \mathbf{b}$ may not have a solution. Note that, from the properties of the determinants:

$$\det(A) = \det(L)\det(U) = \prod_{i=1} nl_{ii} \prod_{i=1} nu_{ii}$$

, where we have used the fact that the determinant of a triangular matrix is the product of its diagonal. Hence, if at least one of the diagonal elements of U and L is zero, A is non-singular hence not invertible, hence the system is not uniquely solvable (see Note 3). This is the case of Exercise 2.16 where we see that $u_{33} = 0$

Consider

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 3 & 3 & 9 \\ 3 & 3 & 5 \end{bmatrix}$$

In Exercise 2.22 you will show that A can be written as $A = LU$ with

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 1 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 9/2 & 15/2 \\ 0 & 0 & -4 \end{bmatrix}$$

Use this factorisation to find the solution to $A\mathbf{x} = \mathbf{b}$ with $\mathbf{b} = \begin{bmatrix} 2 \\ 6 \\ 6 \end{bmatrix}$.

Solution

Following the steps from Section 2.3.1. We first solve $L\mathbf{y} = \mathbf{b}$ via forward substitution:

$$\begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 6 \end{bmatrix}$$

i.e.

$$\begin{aligned} y_1 &= 2 \\ 3/2y_1 + y_2 &= 6 \implies y_2 = 6 - (3/2)y_1 = 6 - (3/2)2 = 3 \\ 3/2y_1 + y_2 + y_3 &= 6 \implies y_3 = 6 - y_2 - (3/2)y_1 = 6 - 3 - (3/2)2 = 0 \end{aligned}$$

Now we solve for $\mathbf{x}$, $U\mathbf{x} = \mathbf{y}$ via backward substitution:

$$\begin{bmatrix} 2 & -1 & 1 \\ 0 & 9/2 & 15/2 \\ 0 & 0 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 0 \end{bmatrix}$$

i.e.

$$\begin{aligned} -4x_3 = 0 &\implies x_3 = 0 \\ 9/2x_2 + 15/2x_3 = 3 &\implies x_2 = (2/9)(3 - (15/2)0) = 2/3 \\ 2x_1 - x_2 + x_3 = 2 &\implies x_1 = 1/2(2 - x_3 + x_2) = 1/2(2 + 2/3) = 4/3 \end{aligned}$$

We can verify our solution using the function `forward_substitution` that you will write in Important 18 as well as the backward substitution from Listing 2.2. Note the latter takes an augmented matrix as input so here we consider the augmented matrix $[U|\mathbf{y}]$.

```
L = np.array([[1, 0, 0],[3/2, 1, 0],[3/2, 1, 1]], dtype=float)
U = np.array([[2, -1, 1],[0, 9/2, 15/2],[0, 0, -4]], dtype=float)

b = np.array([[2], [6], [6]], dtype=float)
y = forward_substitution(L, b)
print("variable y:", y)

U_y=np.hstack((U,y))
x = backward_substitution(U_y)
print("Solution x:", x)

print("verify (LU)x-b=0:", L @ U @ x - b )
```

```
variable y: [[2.]
 [3.]
 [0.]]
Solution x: [[ 1.33333333]
 [ 0.66666667]
 [-0.        ]]
verify (LU)x-b=0: [[0.]
 [0.]
 [0.]]
```

2.3.2 Doolittle's LU Factorisation

In the previous problems we used a specific type of matrix L that has ones on its diagonal. Note that in general, if we want to conduct a factorisation with L and U from Figure 2.9. We may also follow the same procedure matching the n^2 each entries in the equation $A = LU$:

$$A = LU \implies a_{ij} = a_{ij} = (LU)_{ij} = l_{i1}u_{1j} + l_{i2}u_{2j} + \dots l_{in}u_{nj}$$

since L and U are lower and upper triangular we have

$$a_{ij} = l_{i1}u_{1j} + l_{i2}u_{2j} + \dots + l_{ii}u_{ij} + \overbrace{l_{i,i+1}u_{i+1,j}}^{=0} + \dots \overbrace{l_{in}u_{nj}}^{=0} \qquad \text{if} \quad i \le j$$

and

$$a_{ij} = l_{i1}u_{1j} + l_{i2}u_{2j} + \dots + l_{ij}u_{jj} + \overbrace{l_{i,j+1}u_{j+1,j}}^{=0} + \dots \overbrace{l_{in}u_{nj}}^{=0} \qquad \text{if} \quad i \ge j$$

which we can write as

$$a_{ij} = \sum_{k=1}^{\min\{i,j\}} l_{ik}u_{kj} \tag{2.35}$$

Note that if we wanted to solve Equation 2.35 for $i = 1$ and $j = 1, \dots, n$ (i.e. to match the the first row of $A = LU$), we need

$$a_{11} = l_{11}u_{11}, \quad a_{12} = l_{11}u_{12}, \dots, a_{1n} = l_{11}u_{1n} \tag{2.36}$$

We are free to select l_{11} and u_{11} provided $a_{11} = l_{11}u_{11}$. Suppose we specify $l_{11} = 1$ then

$$\begin{aligned} u_{11} &= a_{11}, \\ u_{12} &= a_{12}, \\ \vdots &= \vdots \\ u_{1n} &= a_{1n}. \end{aligned} \tag{2.37}$$

This gives us the first row of U. We now match the first column of $A = LU$, so we consider Equation 2.35 for $j = 1$ and $i = 2, \dots, n$:

$$a_{i1} = \sum_{k=1}^{1} l_{ik}u_{k1} = l_{i1}u_{11} \implies l_{i1} = a_{i1}/u_{11} \tag{2.38}$$

Thus Equation 2.38 delivers the first column of L. Note that we start $i = 2$ since for $i = 1$ we know $l_{11} = 1$.

We now consider $i = 2$ and $j = 2, \dots n$ in Equation 2.35 to match the second row of $A = LU$:

$$\begin{aligned} a_{22} &= l_{21}u_{12} + l_{22}u_{22} \\ a_{23} &= l_{21}u_{13} + l_{22}u_{23} \\ \vdots &= \ \vdots \\ a_{2n} &= l_{21}u_{1n} + l_{22}u_{2n} \end{aligned} \implies \begin{aligned} l_{22}u_{22} &= a_{22} - l_{21}u_{12} \\ l_{22}u_{23} &= a_{23} - l_{21}u_{13} \\ \vdots &= \ \vdots \\ l_{22}u_{2n} &= a_{2n} - l_{21}u_{1n} \end{aligned} \tag{2.39}$$

Notice above we start at $j = 2$ since $j = 1$ was already included in Equation 2.38.

From Equation 2.39 we observe that we can choose $l_{22}u_{22}$ so long $l_{22}u_{22} = a_{22} - l_{21}u_{12}$. If we select $l_{22} = 1$, then

$$\begin{aligned} u_{22} &= a_{22} - l_{21}u_{12} \\ u_{23} &= a_{23} - l_{21}u_{13} \\ \vdots &= \ \vdots \\ u_{2n} &= a_{2n} - l_{21}u_{1n} \end{aligned}$$

and so we have the second row of U. We now compute the second column of L by matching the second column of $A = LU$: take $j = 2$ in Equation 2.35 and $i = 3, \dots n$:

$$a_{i2} = \sum_{m=1}^{\min\{i,2\}} l_{im}u_{m2} = l_{i1}u_{12} + l_{i2}u_{22} \implies l_{i2} = (a_{i2} - l_{i1}u_{12})/u_{22}$$

We have now computed the first and second row of U as well as the first and second columns of L. If we continue the process, using $l_{kk} = 1$, at the kth step we match the k-th row of $A = LU$, i.e. we use Equation 2.35 for $i = k$ and $j = k, \dots n$: so

$$a_{kj} = \sum_{m=1}^{\min\{k,j\}} l_{km}u_{mj} = \sum_{m=1}^{k} l_{km}u_{mj} = \sum_{m=1}^{k-1} l_{km}u_{mj} + \underbrace{l_{kk}}_{=1} u_{kj} \implies u_{kj} = a_{kj} - \sum_{m=1}^{k-1} l_{km}u_{mj} \tag{2.40}$$

so we obtained the non-zeros entries of the k-th row of U, then we proceed to match the k-th column of $A = LU$. Setting $j = k$ in Equation 2.35 and $i = k+1, \dots, n$ we have:

$$a_{ik} = \sum_{m=1}^{\min\{i,k\}} l_{im}u_{mk} = \sum_{m=1}^{k} l_{im}u_{mk} = \sum_{m=1}^{k-1} l_{im}u_{mk} + l_{ik}u_{kk} \implies l_{ik} = \Big(a_{ik} - \sum_{m=1}^{k-1} l_{im}u_{mk}\Big)/u_{kk} \tag{2.41}$$

2.3.3 Pseudocode and python implementation

We have derived a direct approach to obtained Doolitle's LU factorisation which we summarise in Algorithm 2.6 . Doolittle's method is the one we used to solve Exercise 2.16. This method specifies the diagonal of L, $l_{11} = l_{22} = \dots l_{nn} = 1$ and so the resulting matrix L, called **lower unit triangular** is of the form:

$$L = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ \vdots & & \ddots & 0 & \\ \vdots & & & \ddots & \\ l_{n1} & & & & 1 \end{bmatrix} \qquad U = \begin{bmatrix} u_{11} & u_{12} & \dots & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & \ddots & & \\ & 0 & & \ddots & \\ & & & & u_{nn} \end{bmatrix}$$

Complexity and Pivoting

Using the techniques from Section 2.1.4 we can count the number of operations involved to obtain the complexity of Doolittle's method. However, as we will see in Section 2.3.5, Doolittle's method is equivalent to Gaussian elimination and so the complexity is also $\mathcal{O}(n^3)$. Similarly, it is worth noticing that, in general, at some step k the term u_{kk} in Equation 2.41 may be zero or very small resulting in loss of accuracy. Pivoting strategies can also be applied in those cases.

```
A = np.array([[4, 2, 7], [3, 5, -6],[1, -3, 2]], dtype=float)
L, U = doolittle(A)

print("Matrix A:")
print(A)
print("\nLower triangular matrix L:")
print(L)
print("\nUpper triangular matrix U:")
print(U)

# Verifying A = L * U
print("\nVerification (L * U):")
print(L @ U)
```

```
Matrix A:
[[ 4.  2.  7.]
 [ 3.  5. -6.]
 [ 1. -3.  2.]]

Lower triangular matrix L:
[[ 1.    0.    0.  ]
 [ 0.75  1.    0.  ]
 [ 0.25 -1.    1.  ]]

Upper triangular matrix U:
[[  4.     2.     7.  ]
 [  0.     3.5  -11.25]
 [  0.     0.   -11.  ]]

Verification (L * U):
[[ 4.  2.  7.]
 [ 3.  5. -6.]
 [ 1. -3.  2.]]
```

Note

Note that in Listing 2.9 we do not separate the cases $k = 1$ and $k = 2, \ldots n$ as we did in the pseudocode from Algorithm 2.6 . The reason of that is in the pseudocode we have the expression

$$u_{kj} = a_{kj} - \sum_{m=1}^{k-1} l_{km} u_{mj}$$

from which the sum is not defined for $k = 1$. However, when we compute the summation via the `for m in range(k):` loop, the case `k=0` gives `range(0)` which is an empty list and hence the summation is not computed

```
print (list(range(0)) )
```

```
[]
```

so for $k = 1$ the code computes $u_{1j} = a_{1j}$ which is what we need. Similarly for the term

$$l_{ik} = \Big(a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk}\Big)/u_{kk}$$

we obtain $l_{i1} = a_{i1}/u_{11}$.

2.3.4 Other type of factorisations

In the derivation for Doolittle's method in Section 2.3.2 we recognised that we can specify either l_{kk} or u_{kk}. While for Doolittle's method the choice is $l_{kk} = 1$, the choice $u_{kk} = 1$ $(k = 1, \dots, n)$ leads to the so called Crout's method. The resulting factorisation is

$$L = \begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & & \ddots \quad 0 & \\ \vdots & & \ddots & \\ l_{n1} & & & l_{nn} \end{bmatrix} \qquad U = \begin{bmatrix} 1 & u_{12} & \dots & \dots & u_{1n} \\ & 1 & u_{23} & \dots & u_{2n} \\ & & \ddots & & \\ & 0 & & \ddots & \\ & & & & 1 \end{bmatrix}$$

where U is an **upper unit triangular** matrix.

2.3.5 Doolittle's factorisation from Gaussian elimination

Let us define $A^{(1)} = A$ and perform the first step of Gaussian elimination on matrix $A^{(1)}$ noticing that here we do not augment matrix A with $\mathbf{b}$. This is shown in Figure 2.10.

$$\begin{array}{r} E_1: \\ E_2: \\ \vdots \\ E_n: \end{array} \overbrace{\begin{bmatrix} a_{11} & a_{12} \cdots\cdots & a_{1n} \\ a_{21} & a_{22} \cdots\cdots & a_{2n} \\ \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} \cdots\cdots & a_{nn} \end{bmatrix}}^{A^{(1)} \equiv A} \sim \begin{array}{r} E_1 \to E_1 \\ E_2 - m_{21}E_1 \to E_2 \\ E_3 - m_{31}E_1 \to E_3 \\ \vdots \\ E_n - m_{n1}E_1 \to E_n \end{array} \overbrace{\begin{bmatrix} a_{11} & a_{12} \cdots\cdots & a_{1n} \\ 0 & a_{22}^{(2)} \cdots\cdots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} \cdots\cdots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots \\ 0 & a_{n2}^{(2)} \cdots\cdots & a_{nn}^{(2)} \end{bmatrix}}^{A^{(2)}}$$

Figure 2.10: Gaussian Elimination

where for $j = 2, \dots, n$ and $k = 2, \dots n$

$$a_{jk}^{(2)} = a_{jk} - m_{j1}a_{1k}, \qquad m_{j1} = a_{j1}/a_{11} \tag{2.42}$$

You may recall from core module that performing row operations on a matrix is equivalent to pre-multiplying the original matrix by a so-called elementary matrix. This applies to our case of reducing $A^{(1)}$ to $A^{(2)}$. More specifically, in Problem **??** you will show that we can write

$$A^{(2)} = M^{(1)}A^{1)}$$

where $M^{(1)}$ is the elementary matrix given in Figure 2.11.

Notice that $M^{(1)}$ is the identity with entries below the diagonal in the first column replaced by the negative of the multipliers $m_{j1} = a_{j1}/a_{11}$ $(j = 2, \dots, n)$.

Furthermore, let us define $\mathbf{b}^{(1)} = \mathbf{b}$ and note that, since $A^{(1)}x = \mathbf{b}^{(1)}$, then

$$A^{(2)}x = M^{(1)}A^{(1)}x = M^{(1)}\mathbf{b} = M^{(1)}\mathbf{b}^{(1)} = \mathbf{b}^{(2)}$$

Similarly, we can construct $M^{(2)}$ in the second step of Gaussian elimination as shown in Figure 2.12 where $m_{j2} = a_{j2}^{(2)}/a_{22}^{(2)}$ $(j = 3, \dots, n)$.

Note that matrix $M^{(2)}$ is the identity matrix but with entries below the diagonal in the second column replaced by $-m_{j2}$ (for $j = 3, \dots, n$).

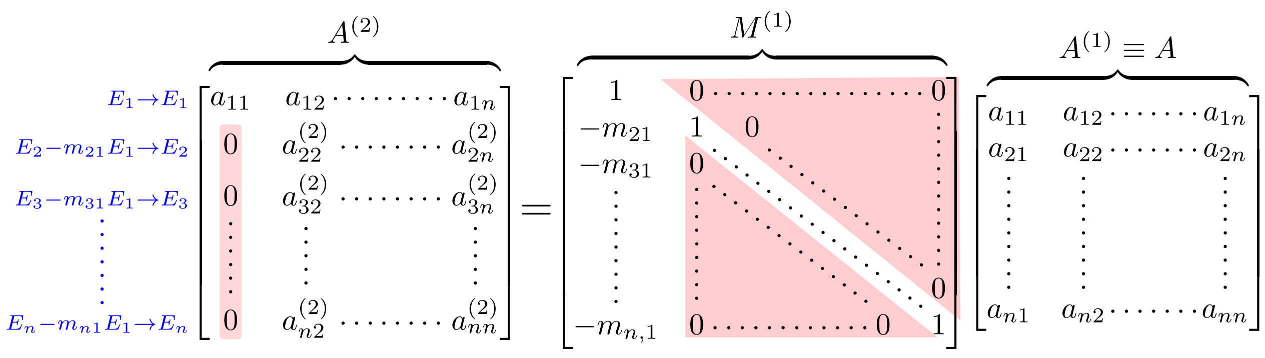

$A^{(2)}$
$M^{(1)}$
$A^{(1)} \equiv A$
$E_1 \to E_1$
$E_2 - m_{21}E_1 \to E_2$
$E_3 - m_{31}E_1 \to E_3$
$E_n - m_{n1}E_1 \to E_n$

Figure 2.11: Gaussian Elimination

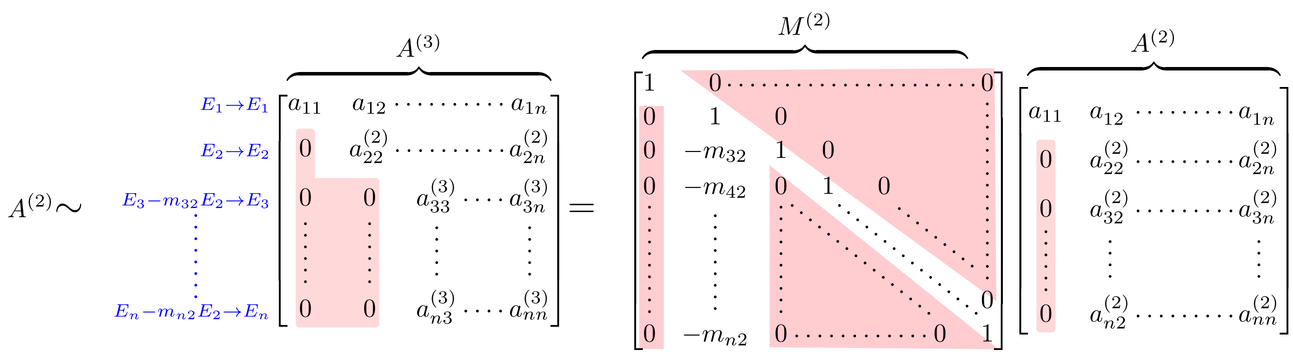


Figure 2.12: Gaussian Elimination

Therefore,

$$A^{(3)}x = M^{(2)}A^{(2)}x = M^{(2)}M^{(1)}A^{(1)}x = M^{(2)}M^{(1)}\mathbf{b}^{(1)} =: \mathbf{b}^{(3)}.$$

Supposed that a step k, we have already built $A^{(k)} = b^{(k)}$, to obtain $A^{(k+1)}$ we construct the matrix $M^{(k)}$ given in Figure 2.13.

Notice that

$$A^{(k+1)}x = M^{(k)}A^{(k)}x = M^{(k)} \cdots M^{(1)}Ax = M^{(k)}\mathbf{b}^{(k)} = \mathbf{b}^{(k+1)} = M^{(k)} \cdots M^{(1)}\mathbf{b}.$$

When we complete the process we form system $A^{(n)}\mathbf{x} = \mathbf{b}^n$ where $A^{(n)}$ is the upper triangular matrix shown in Figure 2.14 which can be represented by

$$A^{(n)} = M^{(n-1)}M^{(n-2)} \cdots M^{(1)}A. \tag{2.43}$$

The process above provides us with the $U = A^{(n)}$ matrix in the LU factorisation of A. Let us know find the corresponding lower-triangular matrix L. To this end, let us note that matrix $M^{(k)}$ (see Figure 2.14) is invertible and given in Figure 2.15.

Te invertibility of $M^{(k)}$ comes from the fact that we transform $A^{(k)}$ to $A^{(k+1)}$ via $E_j - m_{j,k}E_k \to E_j$ (for $j = k+1, \dots n$) so we can then revert such a transformation by $E_j + m_{j,k}E_k \to E_j$

Let us multiply Equation 2.43 by $L^{(n-1)}$ and note that:

$$L^{(n-1)}A^{(n)} = \overbrace{L^{(n-1)}M^{(n-1)}}^{I} M^{(n-2)} \cdots M^{(1)}A,$$

and the last expression by $L^{(n-2)}$

$$L^{(n-2)}L^{(n-1)}A^{(n)} = \overbrace{L^{(n-2)}M^{(n-2)}}^{I} M^{(n-3)} \cdots M^{(1)}A,$$

and continuing the process:

$$L^{(1)} \cdots L^{(n-2)}L^{(n-1)}A^{(n)} = A$$

We define

$$U := A^{(n)} \qquad L := L^{(1)} \cdots L^{(n-2)}L^{(n-1)} \tag{2.44}$$

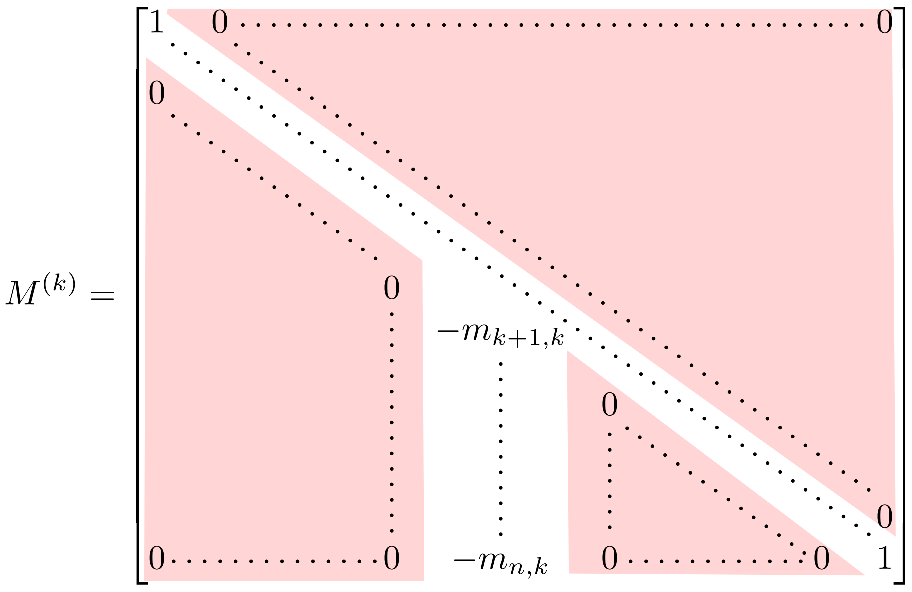


Figure 2.13: Gaussian Elimination

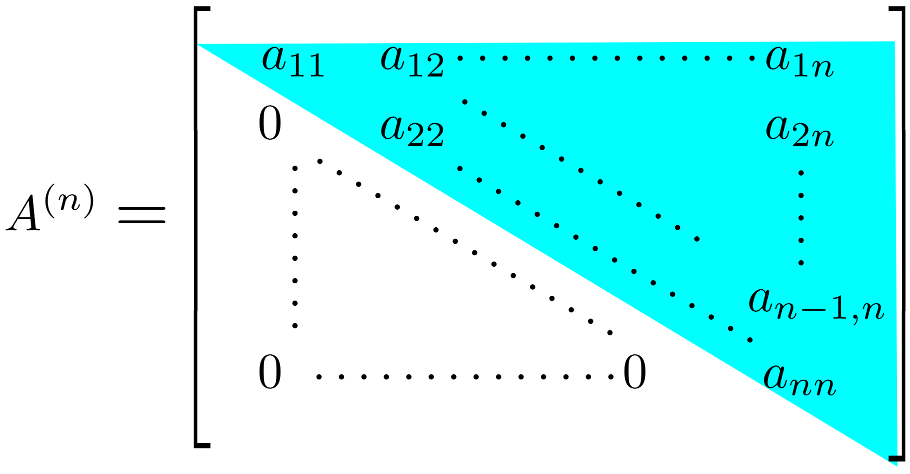


Figure 2.14: Gaussian Elimination

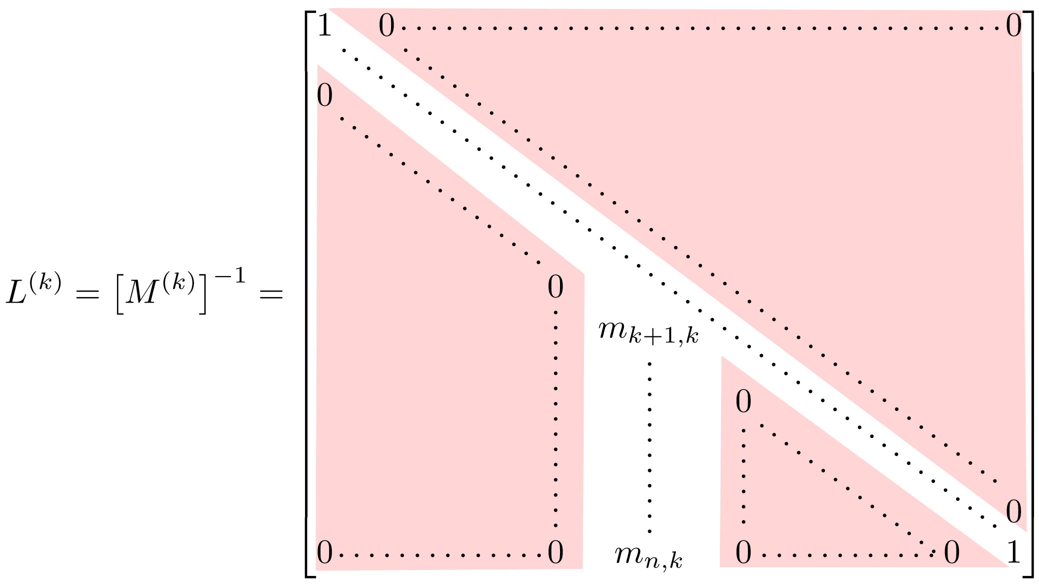


Figure 2.15: Gaussian Elimination

and so

$$LU = A.$$

The above development can be summarised in the following:

Theorem 2.1. *If Gaussian elimination can be performed on the linear system $A\boldsymbol{x} = \boldsymbol{b}$ without row interchanges, then the matrix A can be factored into the product of a lower-triangular matrix L and an upper triangular matrix U, that is $A = LU$, where $m_{ji} = a_{ji}^{(i)}/a_{ii}^{(i)}$,*

$$U = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & \dots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & & \vdots \\ \vdots & 0 & \ddots & \ddots & a_{n-1,n}^{(n-1)} \\ 0 & \dots & \dots & 0 & a_{nn}^{(n)} \end{bmatrix} \quad \textit{and}$$
$$L = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ m_{21} & 1 & \dots & \dots & 0 \\ \vdots & \ddots & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ m_{n,1} & \dots & \dots & m_{n,n-1} & 1 \end{bmatrix} \tag{2.45}$$

Exercise 2.17. Use Theorem **??** to determine the LU factorization for $A = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 0 & -1 & -1 \end{bmatrix}$.

Solution

Note that $m_{21} = a_{21}/a_{11} = 2/1 = 2$ and $m_{31} = a_{31}/a_{11} = 0/1 = 0$ so using $E_2 - m_{21}E_1 \to E_2$ and $E_3 - m_{31}E_1 \to E_3$we have

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 0 & -1 & -1 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

Note that $m_{32} = a_{32}/a_{22} = -1/-1 = 1$ so $E_3 - m_{32}E_2 \to E_3$

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & -1 & -1 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

Hence by Theorem 2.1 we have $L = \begin{bmatrix} 1 & 0 & 0 \\ \overbrace{2}^{m_{21}} & 1 & 0 \\ \overbrace{0}^{m31} & \overbrace{1}^{m_{32}} & 1 \end{bmatrix}$ and $U = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 0 \end{bmatrix}$

Theorem Theorem 2.1 is thus also useful computationally, we can use our Gaussian elmination algorithm (making sure we avoid row interchanges) to compute Doolittle's factorisation. This is done in (**comp-fact-4?**).

2.3.6 Permutation Matrices

One of the limitations of the previous theorem as well as Algorithm 2.6 is that, as we know, in many instances we need to perform row interchanges during Gaussian elimination to solve a linear system. As we mentioned in Note 4, Gaussian elimination can be applued to solve any linear system for a non-singular matrix but row interchanges may be needed. To address the limination of the previous factorisation algorithms, we introduce the following definition.

An $n \times n$ permutation matrix $P = [p_{i,j}]$ is a matrix obtained by rearranging the rows of the $n \times n$ identity matrix

$$I_n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & 0 \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

A permutation matrix has exactly one nonzero entry equal to 1 in each row and each column.

Exercise 2.18. Let A an arbitrary 3×3 matrix and P the permutation matrix given by

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Show that PA is the same as interchanging the second a third row of A.

Solution

The solution is obtained directly from the multiplication below:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

Exercise 2.19. Consider again

$$B = \begin{bmatrix} 0 & -1 & -1 \\ 2 & 1 & -1 \\ 1 & 1 & 0 \end{bmatrix}$$

Find the permutation matrix P so that $PB = \overbrace{\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 0 & -1 & -1 \end{bmatrix}}^{A}$, hence obtain the factorisation $B = P^{-1}LU$

Solution

Note that if we define P by

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix},$$

then $PB = A$. We already know that $A = LU$ then

$$PB = LU \implies B = P^{-1}LU = P^{T}LU$$

Exercise 2.19 is an example of a more general result, namely that for any non-singular matrix A there is a permutation matrix P such that

$$PA = LU \tag{2.46}$$

where L and U are lower unit triangular and upper triangular matrices. In other words, we can always obtain the Doolittle's factorisation of a permuted version of A. This permutation matrix is such that no row interchanges are need in Gaussian elimination. Thus if we need those interchanges before hand, we can contruct P accordingly and then compute PA without any further row interchanges. Using the fact that $P^t = P^{-1}$, Equation 2.46 can be expressed as the so-called PLU-factorisation $A = P^tLU$.

Let us now discuss this in more detail. For simplicity we will consider a 4×4 matrix A. Suppose that we want to conduct the PLU factorisation of A and that row interchanges are needed at each step of the Gaussian elimnation algorithm. We start with P_1 and apply the first step of Gaussian elimination to $P^{(1)}A$ so we obtain

$$A^{(2)} = M^{(1)}(P^{(1)}A)$$

which in terms of the inverse of $M^{(1)}$ is

$$L^{(1)}A^{(2)} = P^{(1)}A$$

Note that $A^{(2)}$ is of the form shown in Figure **??**. Suppose we need to do row interchanges again now on $A^{(2)}$. Notice that we can construct the corresponding permutation matrix $P^{(2)}$ and then multiply this in XXX:

$$P^{(2)}(L^{(1)}A^{(2)}) = P^{(2)}P_1A$$

It is not difficult to see that the left hand side can be expressed as

$$P^{(2)}(L^{(1)}A^{(2)}) = \tilde{L}^{(1)}\tilde{A}^{(2)}$$

where $\tilde{L}^{(1)}$ has the same form as $L^{(1)}$ but with the first column swapped according to $P^{(2)}$ and $\tilde{A}^{(2)} = P^{(2)}A^{(2)}$ as we wanted. We can combine this into:

$$\tilde{L}^{(1)}\tilde{A}^{(2)} = P^{(2)}P_1 A$$

We can now proceed to the next step of Gaussian elimination on $\tilde{A}^{(2)}$

$$A^{(3)} = M^{(2)}\tilde{A}^{(2)}$$

which, again we write as

$$L^{(2)}A^{(3)} = \tilde{A}^{(2)}$$

and so

$$\tilde{L}^{(1)}L^{(2)}A^{(3)} = \tilde{L}^{(1)}\tilde{A}^{(2)} = P^{(2)}P_1 A$$

We repeat the process, assuming that now $A^{(3)}$ requires row interchanges via $P^{(3)}$. We conduct

$$P^{(3)}(\tilde{L}^{(1)}L^{(2)}A^{(3)}) = P^{(3)}P^{(2)}P^{(1)}A$$

and write the LHS as

$$P^{(3)}(\tilde{L}^{(1)}L^{(2)}) = \tilde{L}^{(2)}\tilde{A}^{(3)}$$

where $\tilde{A}^{(3)} = P^{(3)}A^{(3)}$ and $\tilde{L}^{(2)}$ has the entries of the first two columns below the diagonal of $\tilde{L}^{(1)}L^{(2)}$ swapped according to $P^{(3)}$. Then

$$\tilde{L}^{(2)}\tilde{A}^{(3)} = P^{(3)}P^{(2)}P^{(1)}A$$

we do final step of Gaussian elimination

$$A^{(4)} = M^{(3)}\tilde{A}^{(3)}$$

or

$$L^{(3)}A^{(4)} = \tilde{A}^{(3)}$$

so

$$\tilde{L}^{(2)}L^{(3)}A^{(4)} = \tilde{L}^{(2)}\tilde{A}^{(3)} = P^{(3)}P^{(2)}P^{(1)}A$$

Thus $P = P^{(3)}P^{(2)}P^{(1)}$

$$L = \tilde{L}^{(2)}L^{(3)}$$

and

$$U = A^{(4)}$$

2.3.7 Special Types of Matrices

An $n \times n$ matrix A is said to be **diagonally dominant** if

$$|a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \qquad \forall\ i = 1, \dots n. \tag{2.47}$$

When the strict inequality holds in Equation 2.47, matrix A is called **strictly diagonally dominant**. That is when

$$|a_{ii} > \sum_{j=1, j\neq i}^{n} |a_{ij}| \qquad \forall\ i = 1, \dots n.$$

Exercise 2.20. Consider the matrix

$$A = \begin{bmatrix} 7 & 2 & 0 \\ 3 & 5 & -1 \\ 0 & 5 & -6 \end{bmatrix}.$$

Is A and/or A^t strictly diagonally dominant?

Solution

Note that

$$\begin{aligned}|a_{11}| &= 7 > |a_{12}| + |a_{13}| = 2 + 0 = 2\\ |a_{22}| &= 5 > |a_{21}| + |a_{23}| = 3 + |-1| = 4\\ |a_{33}| &= |-6| = 6 > |a_{13}| + |a_{23}| = 0 + 5 = 5\end{aligned}$$

Thus A is strictly diagonal dominant. However, for

$$A^t = \begin{bmatrix} 7 & 3 & 0 \\ 2 & 5 & 5 \\ 0 & -1 & -6 \end{bmatrix}.$$

Note that

$$|a_{22}| = 5 \le |a_{21}| + |a_{23}| = 2 + 5 = 7$$

so A^t is not diagonally dominant.

Theorem 2.2. *A strictly diagonally dominant matrix A is non-singular (i.e. invertible). Moreover, for such a matrix, Gaussian elimination can be performed without row interchanges to solve $A\boldsymbol{x} = \boldsymbol{b}$ for any arbitrary $\boldsymbol{b} \in \mathbb{R}^n$. Furthermore, the computations are stable with respect to the grow of round-off errors.*

Corollary 2.1. *A strictly diagonally dominant matrix A admits the LU factorisation given in Equation 2.45.*

Let us recall the following definition

Definition 2.2. An $n \times n$ matrix A is symmetric if $A = A^t$. A symmetric matrix is **positive definite** if

$$\mathbf{x}^t A\mathbf{x} > 0, \qquad \forall \mathbf{x} \in \mathbb{R}^n \qquad \mathbf{x} \neq \mathbf{0}$$

Exercise 2.21. Consider the matrix $B = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$ Is B positive-definite?

Solution

B is clearly symmetric i.e. $B = B^t$. Also,

$$\begin{aligned}\mathbf{x}^t B\mathbf{x} &= [x_1, x_2]\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = [x_1, x_2]\begin{bmatrix} 2x_1 - x_2 \\ -x_1 + 2x_2 \end{bmatrix}\\ &= 2x_1^2 - x_1x_2 - x_2x_1 + 2x_2^2\\ &= x_1^2 + x_1^2 - x_1x_2 - x_2x_1 + x_2^2 + x_2^2\\ &= x_1^2 + x_1(x_1 - x_2) + (x_2 - x_1)x_2 + x_2^2\\ &= x_1^2 + x_2^2 + (x_1 - x_2)^2 > 0 \qquad \forall [x_1, x_2] \neq [0, 0]\end{aligned}$$

so B is positive definite.

Theorem 2.3. *A symmetric matrix A is positive definite if and only if Gaussian elimination without row interchanges can be performed to solve $A\boldsymbol{x} = \boldsymbol{b}$ with all pivot elements positive. Moreover, the computations are stable with respect to the grow of round-off errors.*

A symmetric matrix A is positive definite if and only if A can be factored in the form $A = LL^t$ where L is lower triangular with nonzero diagonal entries.

2.3.8 Additional Problems

Exercise 2.22. Consider the matrix

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 3 & 3 & 9 \\ 3 & 3 & 5 \end{bmatrix}$$

Find the LU Doolittle's factorisation of A.

Solution

```
A=np.array([[2,-1,1],[3,3,9],[3,3,5]],dtype=float)
U, L=   doolittle(A)
print(U)
print(L)
```

```
[[1.  0.  0. ]
 [1.5 1.  0. ]
 [1.5 1.  1. ]]
[[ 2.  -1.   1. ]
 [ 0.   4.5  7.5]
 [ 0.   0.  -4. ]]
```

Exercise 2.23. Show $A^{(2)} = M^{(1)}A$, where $M^{(1)}$, A and $A^{(2)}$ are given in Figure 2.11.

Solution

$$M^{(1)}A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} - m_{2,1}a_{11} & a_{22} - m_{2,1}a_{12} & a_{23} - m_{2,1}a_{13} & \cdots & a_{2n} - m_{2,1}a_{1n} \\ a_{31} - m_{3,1}a_{11} & a_{32} - m_{3,1}a_{12} & a_{33} - m_{3,1}a_{13} & \cdots & a_{3n} - m_{3,1}a_{1n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} - m_{n,1}a_{11} & a_{n2} - m_{n,1}a_{12} & a_{n3} - m_{n,1}a_{13} & \cdots & a_{nn} - m_{n,1}a_{1n} \end{bmatrix}$$

From the expression from m_{j1} in Equation 2.42 we have that

$$a_{j1} - m_{j1}a_{11} = a_{j1} - (a_{j1}/a_{11})a_{11} = 0$$

Hence all the elements below a_{11} in the first column are zero. Furthermore, from the definition of $a_{jk}^{(2)}$ in Equation 2.42 we have that

$$M^{(1)}A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ & a_{32}^{(2)} & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} \\ \vdots & \vdots & & & \vdots \\ & a_{n2}^{(2)} & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} \end{bmatrix} = A^{(2)}$$

Convinced yourself that L defined in Equation 2.44 can be expressed as shown in **?@eq-mat-fac-newsec2**.

Solution

$$\overbrace{L^{(1)}}^{[M^{(1)}]^{-1}} \overbrace{L^{(2)}}^{[M^{(2)}]^{-1}} = \begin{bmatrix} 1 & 0 & \dots & \dots & \dots & \dots & 0 \\ m_{21} & 1 & 0 & \dots & \dots & \dots & 0 \\ m_{31} & 0 & 1 & 0 & \dots & \dots & 0 \\ m_{41} & 0 & 0 & 1 & 0 & \dots & 0 \\ & & & & & & \\ \vdots & \vdots & \vdots & & & & \vdots \\ m_{n-1,1} & 0 & 0 & & & \ddots & 0 \\ m_{n1} & 0 & 0 & \dots & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & 1 & 0 & \dots & \dots & \dots & 0 \\ 0 & m_{32} & 1 & 0 & \dots & \dots & 0 \\ 0 & m_{42} & 0 & 1 & 0 & \dots & 0 \\ & & & & & & \\ \vdots & \vdots & & & & & \vdots \\ 0 & m_{n-1,2} & 0 & & & \ddots & 0 \\ 0 & m_{n2} & 0 & \dots & \dots & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & \dots & \dots & \dots & 0 \\ m_{21} & 1 & 0 & \dots & \dots & \dots & 0 \\ m_{31} & m_{32} & 1 & \dots & \dots & \dots & 0 \\ m_{41} & m_{42} & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & & & & & \vdots \\ m_{n-1,1} & m_{n-1,2} & 0 & & & \ddots & 0 \\ m_{n1} & m_{n2} & 0 & \dots & \dots & 0 & 1 \end{bmatrix}$$

Then,

$$
\begin{aligned}
L^{(1)}L^{(2)}\overbrace{L^{(3)}}^{[M^{(3)}]^{-1}} &= \begin{bmatrix} 1 & 0 & 0 & \dots & \dots & \dots & 0 \\ m_{21} & 1 & 0 & \dots & \dots & \dots & 0 \\ m_{31} & m_{32} & 1 & \dots & \dots & \dots & 0 \\ m_{41} & m_{42} & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & & & & & \vdots \\ m_{n-1,1} & m_{n-1,2} & 0 & & & \ddots & 0 \\ m_{n1} & m_{n2} & 0 & \dots & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & 1 & 0 & \dots & \dots & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & \dots & 0 \\ 0 & 0 & m_{43} & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & & & \vdots \\ 0 & 0 & m_{n-1,3} & & & \ddots & 0 \\ 0 & 0 & m_{n,3} & \dots & \dots & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & \dots & \dots & \dots & 0 \\ m_{21} & 1 & 0 & \dots & \dots & \dots & 0 \\ m_{31} & m_{32} & 1 & \dots & \dots & \dots & 0 \\ m_{41} & m_{42} & m_{43} & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & & & \vdots \\ m_{n-1,1} & m_{n-1,2} & m_{n-1,3} & & & \ddots & 0 \\ m_{n1} & m_{n2} & m_{n,3} & \dots & \dots & 0 & 1 \end{bmatrix}
\end{aligned}
$$

Continue the process we arrive at **?@eq-mat-fac-newsec2**.

2.3.9 Computer Activities

! Important 18: Computer Activity

Write a function `forward_substitution` that implements the forward substitution algorithm discussed in Section 2.3.1. You may want to write the pseudocode before you do the python implementation.

! Important 19: Computer Activity

Write a function `Gauss_LU` that computes Doolittle *LU* factorisation via Gaussian elimination. You may want to reuse the codes from Section 2.1.

Listing 2.4 Profiling elementary operations in python.

```
import time
n=10000
x = np.maximum(np.random.rand(n,1),0.01)
y = np.random.rand(n,1)

start_time = time.time()
for i in range(n):
  x[i]+y[i]

end_time = time.time()
print('addition:', end_time-start_time)

start_time = time.time()
for i in range(n):
  x[i]-y[i]

end_time = time.time()
print('subtraction:', end_time-start_time)

start_time = time.time()
for i in range(n):
  x[i]*y[i]

end_time = time.time()
print('multiplication:', end_time-start_time)

start_time = time.time()
for i in range(n):
  y[i]/x[i]

end_time = time.time()
print('division:', end_time-start_time)


start_time = time.time()
for i in range(n):
  (y[i] == x[i])

end_time = time.time()
print('comparisons:', end_time-start_time)
```

```
addition: 0.004142284393310547
subtraction: 0.003991842269897461
multiplication: 0.004498720169067383
division: 0.0038568973541259766
comparisons: 0.004655122756958008
```

Listing 2.5

```python
import numpy as np
import time
import matplotlib.pyplot as plt
import directsolvers as dsolve



sizes = [ 20, 40, 80, 100, 160,250, 320]
times = []

for n in sizes:
    # Generate a random system of equations
    A = np.random.rand(n, n)
    b = np.random.rand(n,1)

    start_time = time.time()

    tildeA = dsolve.naive_gaussian_elimination(A, b)

    end_time = time.time()

    times.append(end_time - start_time)

# Plot the results
fig, ax = plt.subplots()
ax.set_yscale("log")
ax.set_xscale("log")

ax.plot(sizes, times, marker='o', label='(CPU time) Gaussian
 ↪  Elimination',color='blue')
ax.plot(sizes, 1e-6*np.array(sizes)**3, marker='d', label=r'$\propto
 ↪  n^3$',color='red')
ax.plot(sizes, 1e-6*np.array(sizes)**2, marker='d', label=r'$\propto
 ↪  n^2$',color='green')


ax.set_xlabel('Matrix Size (n)')
ax.set_ylabel('Time (seconds)')
ax.legend(fontsize=16, loc ='best')
```

Listing 2.6 Testing complexity of Gaussian elimination

```python
import numpy as np
import time
import matplotlib.pyplot as plt
import directsolvers as dsolve


sizes = [ 20, 40, 80, 100, 160,250, 320]
times = []

for n in sizes:
    # Generate a random system of equations
    A = np.random.rand(n, n)
    b = np.random.rand(n,1)

    start_time = time.time()

    tildeA = dsolve.gaussian_elimination_vectorised(A, b)

    end_time = time.time()

    times.append(end_time - start_time)

# Plot the results
fig, ax = plt.subplots()
ax.set_yscale("log")
ax.set_xscale("log")

ax.plot(sizes, times, marker='o', label='(CPU time) Gaussian
 ↪  Elimination',color='blue')
ax.plot(sizes, 1e-6*np.array(sizes)**3, marker='d', label=r'$\propto
 ↪  n^3$',color='red')
ax.plot(sizes, 1e-6*np.array(sizes)**2, marker='d', label=r'$\propto
 ↪  n^2$',color='green')

ax.set_xlabel('Matrix Size (n)')
ax.set_ylabel('Time (seconds)')
ax.legend(fontsize=16, loc ='best')
```

Algorithm 2.3 Gaussian Elimination with standard pivoting

```
Input: $n \times n$ matrix $A$. $n \times 1$ vector $b$.
procedure GaussianElimination($A$,$b$)
    Define augmented matrix $\tilde{A} = [A \,|\, b]$. Denote by $a_{ij}$ the entries of $\tilde{A}$.
    for $i = 1$ to $n - 1$ do
        if $a_{ii} = 0$ then
            Find smallest $p$ with $a_{pi} \neq 0$ then $E_p \leftrightarrow E_i$
            if $p$ does not exists then
                Terminate. Output: the system does not have unique solution
            end if
        end if
        for $j = i + 1$ to $n$ do
            Compute $m_{ji} = a_{ji}/a_{ii}$
            Set $a_{ji} = 0$
            for $k = i + 1$ to $n + 1$ do
                $a_{jk}= a_{jk} - m_{ji} a_{ik}$
            end for
        end for
    end for
end procedure
Output: $\tilde{A}$ in reduced form.
```

Listing 2.7 Implementation of Gaussian elimination with standard pivoting

```python
import numpy as np

def gaussian_elimination(A,b):
    """
    Returns numpy array with the augmented matrix tildeA obtained by
    Gaussian elimination starting from the augmented matrix [A b].
    This function performs row interchanging if zero pivot is found.

    Parameters (Inputs)
    ----------
    A : numpy.ndarray of shape (n,n)
        Array representing the square matrix A.
    b : numpy.ndarray of shape (n,1)
        Array representing the column vector b.

    Returns
    -------
    tildeA : numpy.ndarray of shape (n,n+1)
        Array representing the augmented matrix tildeA.
    """

    # Create the initial augmented matrix
    tildeA=np.hstack((A,b))

    #number of rows:
    n = np.shape(tildeA)[0]

    # Start Gaussian Elimination
    for i in range(n-1):

        #Check if pivot is found
        if tildeA[i, i] == 0:
            for p in np.arange(i + 1, n):
                if tildeA[p, i] != 0:
                    # Swap the rows
                    tildeA[[i, p]] = tildeA[[p, i]]
                   print('zero pivot found, swapping row ',i+1,' with row ',p+1)
                    break
            else:
                raise ValueError("The system does not have unique solution.")
                 ↪



        for j in np.arange(i+1,n):
            #compute multiplier
            m = tildeA[j,i] / tildeA[i,i]
            #make zeros below pivot
            tildeA[j,i] = 0
            #update jth row (only the non-zero elements)
            for k in np.arange(i+1,n+1):
                tildeA[j,k] = tildeA[j,k] - m * tildeA[i,k]


    return tildeA


if __name__ == "__main__":
  A=np.array([[0,1,1],[1,1,1],[1,2,1]],dtype=float)
  b=np.array([[1],[1],[4]],dtype=float)
  tildeA=   gaussian_elimination(A, b)
  print(f"Reduced matrix:\ntildeA =\n{tildeA}")
```

```
zero pivot found, swapping row  1  with row  2
Reduced matrix:
```

Algorithm 2.4 Gaussian Elimination with partial pivoting

```
1: **Input:** $n \times n$ matrix $A$. $n \times 1$ vector $\mathbf{b}$.
2: **procedure** GaussianEliminationPartialPivoting($A$,$\mathbf{b}$)
3:     Define augmented matrix $\tilde{A} = [A\ \mathbf{b}]$.
4:     Denote by $a_{ij}$ the entries of $\tilde{A}$.
5:     **for** $i = 1$ **to** $n - 1$ **do**
6:         Find smallest $p$ with $i \leq p \leq n$ such that $|a_{pi}| = \max_{i \leq j \leq n} |a_{ji}|$
7:         **if** $a_{pi} = 0$ **then**
8:             Terminate. **Output:** the system does not have unique solution
9:         **else**
10:            Swap rows: $E_p \leftrightarrow E_i$
11:        **end if**
12:        **for** $j = i + 1$ **to** $n$ **do**
13:            Compute $m_{ji} = a_{ji}/a_{ii}$
14:            Set $a_{ji} = 0$
15:            **for** $k = i + 1$ **to** $n + 1$ **do**
16:                $a_{jk} = a_{jk} - m_{ji}a_{ik}$
17:            **end for**
18:        **end for**
19:    **end for**
20: **end procedure**
21: **Output:** $\tilde{A}$ in reduced form.
```

Listing 2.8 Implementation of Gaussian elimination with partial pivoting

```python
def gaussian_elimination_partial_pivoting(A, b):
    """
    Returns an array representing the augmented matrix tildeA arrived at by
    starting from the augmented matrix [A b] and performing forward
    elimination, with partial pivoting, until all of the
    entries below the main diagonal are 0, assuming the
    matrix A is such that this can be done.

    Parameters
    ----------
    A : numpy.ndarray of shape (n,n)
        Array representing the square matrix A.
    b : numpy.ndarray of shape (n,1)
        Array representing the column vector b.

    Returns
    -------
    tildeA : numpy.ndarray of shape (n,n+1)
        Array representing the augmented matrix tildeA.
    """

    tildeA = np.hstack((A, b))
    n = np.shape(tildeA)[0]

    # Perform Gaussian elimination
    for i in range(n):
        # Partial pivoting:

        #First:find the row with the largest pivot
        max_a=np.abs(tildeA[i, i])
        max_row=i
        for k in np.arange(i + 1, n):
            abs_Aki=np.abs(tildeA[k, i])
            if abs_Aki >max_a:
                max_row=k
                max_a=abs_Aki
        #Second: swap rows with largest pivot
        if i != max_row:
            tildeA[[i, max_row]] = tildeA[[max_row, i]]
            print("swaping row ", i+1," with row ", max_row+1)

        # Eliminate entries below the pivot
        for j in np.arange(i+1,n):
            m = tildeA[j,i] / tildeA[i,i]
            tildeA[j,i] = 0
            tildeA[j, i+1:] -= m * tildeA[i, i+1:]

    return tildeA

if __name__ == "__main__":
  A=np.array([[1e-16,1,1],[1,1,1],[1,2,1]])
  b=np.array([[1],[1],[4]])
  tildeA=   gaussian_elimination_partial_pivoting(A, b)
  x=backward_substitution(tildeA)
  print(tildeA)
  print(x)
```

```
swaping row  1  with row  2
swaping row  2  with row  3
[[ 1.   1.   1.   1.]
 [ 0.   1.   0.   3.]
 [ 0.   0.   1. -2.]]
[[-2.22044605e-16]
 [ 3.00000000e+00]
 [-2.00000000e+00]]
```

Algorithm 2.5 Gaussian elimination with scaled pivoting

```
1: **Input:** $n \times n$ matrix $A$. $n \times 1$ vector $\mathbf{b}$.
2: **procedure** GaussianEliminationScaledPivoting($A$,$\mathbf{b}$)
3:     Define augmented matrix $\tilde{A} = [A\ \mathbf{b}]$.
4:     Denote by $a_{ij}$ the entries of $\tilde{A}$.
5:     Compute scaled factors:
6:     **for** $i = 1$ **to** $n$ **do**
7:         Compute $s_i = \max_{1 \le j \le n} |a_{ij}|$ $(1 \le i \le n)$
8:         **if** $s_i = 0$ **then**
9:             Terminate. **Output:** the system does not have unique solution
10:        **end if**
11:    **end for**
12:    **for** $i = 1$ **to** $n - 1$ **do**
13:        Find smallest $p$ with $i \le p \le n$ such that $\frac{|a_{pi}|}{s_p} = \max_{i \le j \le n} \frac{|a_{ji}|}{s_j}$
14:        **if** $p \neq i$ **then**
15:            Swap rows: $E_p \leftrightarrow E_i$
16:            Swap scaling factors: $s_p \leftrightarrow s_i$
17:        **end if**
18:        **for** $j = i + 1$ **to** $n$ **do**
19:            Compute $m_{ji} = a_{ji}/a_{ii}$
20:            Set $a_{ji} = 0$
21:            **for** $k = i + 1$ **to** $n + 1$ **do**
22:                $a_{jk} = a_{jk} - m_{ji} a_{ik}$
23:            **end for**
24:        **end for**
25:    **end for**
26: **end procedure**
27: **Output:** $\tilde{A}$ in reduced form.
```

Algorithm 2.6 Doolittle's factorisation

```
1: **Input:** $n \times n$ matrix $A$
2: **procedure** DoolittleFactorisation($A$)
3:     Set $l_{ii} = 0$ for $i = 1 \to n$.
4:     Compute $u_{1k} = a_{1k}$ for $k = 1, \dots, n$.
5:     Compute $l_{i1} = a_{i1}/u_{11}$ for $i = 2 \to n$.
6:     **for** $k = 2$ **to** $n$ **do**
7:         **for** $j = k$ **to** $n$ **do**
8:             Compute $u_{kj} = a_{kj} - \sum_{m=1}^{k-1} l_{km} u_{mj}$
9:         **end for**
10:        **for** $i = k + 1$ **to** $n$ **do**
11:            $l_{ik} = \left(a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk}\right)/u_{kk}$
12:        **end for**
13:    **end for**
14: **end procedure**
15: **Output:** $L$ and $U$ factors of $A$
```

Listing 2.9 Implementation of Doolittle's Method

```python
import numpy as np

def doolittle(A):
    """
    Perform Doolittle Factorization on matrix A.
    Decomposes A into L (lower triangular) and U (upper triangular) such that A
 ↪  = L * U.

    Parameters:
    A (numpy.ndarray): The input square matrix

    Returns:
    L (numpy.ndarray): Lower triangular matrix
    U (numpy.ndarray): Upper triangular matrix
    """
    n = A.shape[0]
    L = np.identity(n)
    U = np.zeros((n, n))

    for k in range(n):
        # Constructing the U matrix
        for j in range(k, n):
            s=0
            for m in range(k):
              s+= L[k, m] * U[m, j]
            U[k, j] = A[k, j] - s

        # Constructing the L matrix
        for j in range(k + 1, n):
            s=0
            for m in range(k):
              s+= L[j, m] * U[m, k]

            L[j, k] = (A[j, k] - s ) / U[k, k]

    return L, U
```

Chapter 3

Iterative Techniques

As with direct methods, the aim is to solve the linear system of n equations:

$$\begin{aligned}
E_1 : &\quad a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
E_2 : &\quad a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\
\vdots &\quad\quad \vdots \\
E_n : &\quad a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n
\end{aligned} \tag{3.1}$$

While with Direct Methods we compute the exact solution $\mathbf{x} = (x_1, \dots, x_n)$ of the system above, Iterative Methods provide an approximation to $\mathbf{x}$. The advantage, however, is that their computational cost is lower than direct methods.

3.1 Jacobi iterative method

To introduce Jacobi iterative method let us look at the ith equation of the linear system from Equation 3.1:

$$E_i : \qquad a_{i1}x_1 + \cdots + a_{in}x_n = b_i, \qquad i = 1, \dots, n.$$

Notice that we can write this equation as

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \qquad i = 1, \dots, n$$

or

$$a_{ii}x_i + \sum_{j=1, j\neq i}^{n} a_{ij}x_j = b_i, \qquad i = 1, \dots, n$$

and assuming $a_{ii} \neq 0$:

$$x_i = -\frac{1}{a_{ii}} \sum_{j=1, j\neq 1}^{n} a_{ij}x_j + \frac{b_i}{a_{ii}}, \qquad i = 1, \dots, n \tag{3.2}$$

The idea of **Jacobi's method** is to use Equation 3.2 in the following iterative fashion:

$$x_i^{(k)} = -\frac{1}{a_{ii}} \sum_{j=1, j\neq 1}^{n} a_{ij}x_j^{(k-1)} + \frac{b_i}{a_{ii}}, \qquad i = 1, \dots, n \tag{3.3}$$

where the super-index ${}^{(k)}$ denotes the iteration level. Given an initial guess $\mathbf{x}^{(0)} = [x_1^{(0)}, \dots, x_n^{(0)}]^t$, the aim is to compute $\mathbf{x}^{(k)} = [x_1^{(k)}, \dots, x_n^{(k)}]^t$, for $k = 1, 2, \dots$ using Equation 3.3.

Exercise 3.1. Consider the linear system $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 4 \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{pmatrix} 48 \\ 12 \\ 24 \end{pmatrix}.$$

- Given the initial vector $\mathbf{x}^{(0)} = \begin{pmatrix}1\\1\\1\end{pmatrix}$, use the Jacobi method to obtain $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$.
- Do you think the sequence $\{\mathbf{x}^{(k)}\}_{k=0}^{\infty}$ converges to the solution $\mathbf{x} = \begin{bmatrix}13\\4\\7\end{bmatrix}$?

Solution

$$
\begin{aligned}
x_1^{(1)} &= -\frac{1}{a_{11}}\left[a_{12}x_2^{(0)} + a_{13}x_3^{(0)}\right] + \frac{b_1}{a_{11}} = -\frac{1}{4}\left[(-1)(1) + (0)(1)\right] + \frac{48}{4} = \frac{49}{4} = 12.25\\
x_2^{(1)} &= -\frac{1}{a_{22}}\left[a_{21}x_1^{(0)} + a_{23}x_3^{(0)}\right] + \frac{b_2}{a_{22}} = -\frac{1}{8}\left[(-1)(1) + (-1)(1)\right] + \frac{12}{8} = \frac{14}{8} = 1.75\\
x_3^{(1)} &= -\frac{1}{a_{33}}\left[a_{31}x_1^{(0)} + a_{32}x_2^{(0)}\right] + \frac{b_3}{a_{33}} = -\frac{1}{4}\left[(0)(-1) + (-1)(1)\right] + \frac{24}{4} = \frac{25}{4} = 6.25
\end{aligned}
$$

```python
import numpy as np

A = np.array([[4, -1, 0],
                  [-1, 8, -1],
                  [0, -1,4]], dtype=float)

b = np.array([48, 12, 24], dtype=float)

x0 = np.array([1, 1, 1], dtype=float)


def updateJacobi_simple(A,b,x):

    x_new=np.zeros((3,))
    x_new[0]=1/A[0,0] * ( -( A[0,1]*x[1]+A[0,2]*x[2] ) +b[0])
    x_new[1]=1/A[1,1] * ( -( A[1,0]*x[0]+A[1,2]*x[2] ) +b[1])
    x_new[2]=1/A[2,2] * ( -( A[2,0]*x[0]+A[2,1]*x[1] ) +b[2])
    return x_new

x=updateJacobi_simple(A, b, x0)
print(x)


x=updateJacobi_simple(A, b, x)
print(x)

x=updateJacobi_simple(A, b, x)
print(x)

x=updateJacobi_simple(A, b, x)
print(x)
```

```
[12.25   1.75   6.25]
[12.4375   3.8125   6.4375]
[12.953125   3.859375   6.953125]
[12.96484375   3.98828125   6.96484375]
```

Seems to converge to $\begin{pmatrix}13\\4\\7\end{pmatrix}$

The obvious question now is how/when to stop Jacobi's method. Notice that in contrast to the iterative methods from Chapter 1 where the aim was to approximate a scalar variable, Jacobi's method involves the approximation of a vector. Nonetheless, stopping criteria will require the notion of a distance in $\mathbb{R}^n$ and the convergence of a vector sequence.

3.1.1 Convergence of vectors

Definition 3.1. A vector norm on $\mathbb{R}^n$ is a function, denoted by $\|\cdot\|$, from $\mathbb{R}^n$ into $\mathbb{R}$ with the following properties

1. $\|\mathbf{x}\| \geq 0$ for all $\mathbf{x} \in \mathbb{R}^n$
2. $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = \mathbf{0}$
3. $\|\alpha\mathbf{x}\| = |\alpha| \, \|\mathbf{x}\|$ for all $\alpha \in R$ and $\mathbf{x} \in \mathbb{R}^n$
4. $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ for all $\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^n$

Definition 3.2. The l_2 and l_∞ norms for the vector $\mathbf{x} = (x_1, x_2, \dots, x_n)^t$ are defined by

$$\|\mathbf{x}\|_2 = \left\{ \sum_{i=1}^{n} x_i^2 \right\}^{1/2}$$

and

$$\|\mathbf{x}\|_\infty = \max_{1 \leq i \leq n} |x_i|$$

respectively.

The l_2 norm is nothing but the Eucledian norm induced by by the inner (or dot) product of two vectors. Recall that, for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ the inner product is defined by

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i.$$

and so

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x} \cdot \mathbf{y}}$$

Cauchy–Schwarz inequality

For any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ the following inequality holds:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i \leq \left\{ \sum_{i=1}^{n} x_i^2 \right\}^{1/2} \left\{ \sum_{i=1}^{n} y_i^2 \right\}^{1/2} = \|\mathbf{x}\|_2 \, \|\mathbf{y}\|_2$$

Exercise 3.2. Compute the l_2 and l_∞ norms of the vector $\mathbf{x} = [1, 3, -\pi, 2]^t$.

Solution

$$\|\mathbf{x}\|_2 = \left\{ \sum_{i=1}^{4} x_i^2 \right\}^{1/2} = \sqrt{1^2 + 3^2 + (-\pi)^2 + 2^2} = \sqrt{14 + \pi^2} \approx 4.8856$$

$$\|\mathbf{x}\|_\infty = \max_{1 \leq i \leq 4} |x_i| = \max\left\{|1|, \, |3|, |-\pi| \, , |2|\right\} = \pi$$

```
x = np.array([1.0, 3.0, -np.pi, 2.0])
print(np.linalg.norm( x ) )
print(np.linalg.norm( x , np.inf ) )
import numpy.linalg as LA

print(LA.norm(x))
print(LA.norm(x,np.inf))
```

```
4.885652914512999
3.141592653589793
4.885652914512999
3.141592653589793
```

Theorem 3.1. *For each $\boldsymbol{x} \in \mathbb{R}^n$,*

$$\|\boldsymbol{x}\|_\infty \leq \|\boldsymbol{x}\|_2 \leq \sqrt{n} \, \|\boldsymbol{x}\|_\infty \tag{3.4}$$

Proof. Note that $|x_j|^2 \leq \sum_{i=1}^{n} x_i^2$ for all $j = 1, \dots, n$, thus

$$|x_j| \leq \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2} = \|\mathbf{x}\|_2 .$$

Therefore,

$$\|\mathbf{x}\|_\infty = \max_j |x_j| \leq \|\mathbf{x}\|_2$$

On the other hand

$$\|\mathbf{x}\|_2^2 = \sum_{i=1}^{n} x_i^2 \leq \sum_{i=1}^{n} \Big(\max_j |x_j| \Big)^2 = n \Big(\max_j |x_j| \Big)^2 = n \|\mathbf{x}\|_\infty^2$$

hence $\|\mathbf{x}\|_2 \leq \sqrt{n} \|\mathbf{x}\|_\infty$. □

Definition 3.3. A sequence $\{\mathbf{x}\}_{k=1}^{\infty}$ of vectors in $\mathbb{R}^n$ is said to converge to $\mathbf{x}$ with respect to the norm $\|\cdot\|$ if given $\epsilon > 0$, there exists an integer $N(\epsilon)$ such that

$$\left\| x^{(k)} - \mathbf{x} \right\| < \epsilon \qquad \forall\ k \geq N(\epsilon)$$

Theorem 3.2. *The sequence of vectors $\{\boldsymbol{x}\}_{k=1}^{\infty}$ converges to $\boldsymbol{x} \in \mathbb{R}^n$ with respect to the l_∞ norm if and only if $\lim_{k\to\infty} x_i^{(k)} = x_i$ for each $i = 1, 2, \dots, n$.*

Exercise 3.3. Consider the sequence $\{\mathbf{x}^{(k)}\}$ defined by

$$\mathbf{x}^{(k)} = \begin{pmatrix} (\frac{1}{2})^k \\ 1 + \frac{1}{(k+1)} \\ 13 \end{pmatrix}$$

1. Compute the limit of each component:

$$\lim_{k\to\infty} x_1^{(k)}, \quad \lim_{k\to\infty} x_2^{(k)}, \quad \lim_{k\to\infty} x_3^{(k)}.$$

2. Define $\mathbf{x}$ to be the vector containing these limits. Compute $\left\| \mathbf{x}^{(k)} - \mathbf{x} \right\|_\infty$. What is the limit of this norm as $k \to \infty$?

2.1 Does sequence $\{\mathbf{x}^{(k)}\}$ converge to $\mathbf{x}$ with respect to $\|\cdot\|_\infty$? 2.2 And, how about its convergence with respect to $\|\cdot\|_2$?

Solution

Note that

$$\lim_{k\to\infty} x_1^{(k)} = \lim_{k\to\infty} \left(\frac{1}{2}\right)^k = 0$$
$$\lim_{k\to\infty} x_2^{(k)} = \lim_{k\to\infty} \left[1 + \frac{1}{(k+1)}\right] = 1$$
$$\lim_{k\to\infty} x_3^{(k)} = \lim_{k\to\infty} 13 = 13$$

By Theorem 3.2 $\mathbf{x}^{(k)}$ converges to $\mathbf{x} = \begin{pmatrix} 0 \\ 1 \\ 13 \end{pmatrix}$ with respect to $\|\cdot\|_\infty$. Indeed,

$$\left\| \mathbf{x}^{(k)} - \mathbf{x} \right\| = \left\| \begin{pmatrix} (\frac{1}{2})^k \\ 1 + \frac{1}{(k+1)} \\ 13 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 13 \end{pmatrix} \right\|_\infty = \left\| \begin{pmatrix} (\frac{1}{2})^k \\ \frac{1}{(k+1)} \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 13 \end{pmatrix} \right\|_\infty = \frac{1}{k+1} \to 0$$

as $k \to \infty$ since $k + 1 < 2^k$ for $k > 2$.

Corollary 3.1. *The sequence of vectors $\{\boldsymbol{x}^{(k)}\}_{k=1}^{\infty}$ converges to $\boldsymbol{x} \in \mathbb{R}^n$ with respect to the l_∞ if and only if $\{\boldsymbol{x}^{(k)}\}_{k=1}^{\infty}$ converges to $\boldsymbol{x}$ with respect to the l_2 norm*

Proof. Let $\epsilon > 0$. Since $\{\mathbf{x}^{(k)}\}_{k=1}^{\infty} \to \mathbf{x}$ with respect to the l_∞ norm, there is $N \in \mathbb{N}$ such that

$$\left\|\mathbf{x}^{(k)} - \mathbf{x}\right\|_\infty < \tilde{\epsilon} \equiv \epsilon/\sqrt{n}$$

for all $k > N$. From Equation 3.4 we have

$$\left\|\mathbf{x}^{(k)} - \mathbf{x}\right\|_2 \leq \sqrt{n}\left\|\mathbf{x}^{(k)} - \mathbf{x}\right\|_\infty \leq \sqrt{n}\left(\epsilon/\sqrt{n}\right) = \epsilon$$

for all $k > N$ and so $\{\mathbf{x}^{(k)}\}_{k=1}^{\infty} \to \mathbf{x}$ with respect to the l_2 norm. A similar argument shows the other implication. $\square$

3.1.2 Jacobi's Method in vector form.

Recall that we can write the system of equations from Equation 3.1 as $A\mathbf{x} = \mathbf{b}$ where

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & a_{n-1,n} \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix}$$

Note that we can write

$$A = D - L - U,$$

as folllows:

$$\overbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & a_{n-1,n} \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix}}^{A} =$$

$$\overbrace{\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & a_{nn} \end{bmatrix}}^{D} - \overbrace{\begin{bmatrix} 0 & 0 & \cdots & 0 \\ -a_{21} & 0 & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ -a_{n1} & \cdots & -a_{n,n-1} & 0 \end{bmatrix}}^{L} - \overbrace{\begin{bmatrix} 0 & -a_{21} & \cdots & -a_{1n} \\ \vdots & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -a_{n-1,n} \\ 0 & \cdots & 0 & 0 \end{bmatrix}}^{U}$$

i.e. with

$$D = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & a_{nn} \end{bmatrix}$$

and

$$L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ -a_{21} & 0 & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ -a_{n1} & \cdots & -a_{n,n-1} & 0 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 0 & -a_{21} & \cdots & -a_{1n} \\ \vdots & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -a_{n-1,n} \\ 0 & \cdots & 0 & 0 \end{bmatrix}$$

Then $A\mathbf{x} = \mathbf{b}$ can be written as

$$A\mathbf{x} = \mathbf{b} \Longrightarrow (D - L - U)\mathbf{x} = \mathbf{b} \Longrightarrow D\mathbf{x} = (L + U)\mathbf{x} + \mathbf{b}. \tag{3.5}$$

Note that if $a_{ii} \neq 0$ for all $i = 1, \ldots n$, then D^{-1} exists. In fact,

$$D^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{22}} & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & 0 & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & \frac{1}{a_{nn}} \end{bmatrix}$$

So we can write (Equation 3.5) as

$$\mathbf{x} = \overbrace{D^{-1}(L+U)}^{T}\,\mathbf{x} + \overbrace{D^{-1}\mathbf{b}}^{\mathbf{c}}.$$

Jacobi's method in vector form: Given, $\mathbf{x}^{(0)} \in \mathbb{R}^n$, compute

$$\mathbf{x}^{(k)} = T\mathbf{x}^{(k-1)} + \mathbf{c}$$

where

$$\begin{aligned} T =& D^{-1}(L+U) \\ \mathbf{c} =& D^{-1}\mathbf{b} \end{aligned}$$

assuming that D^{-1} exists.

Exercise 3.4. Consider again the linear system $A\mathbf{x} = \mathbf{b}$ from Exercise 3.1.

1. Determine the matrix $\mathbf{T}$ and vector $\mathbf{c}$ in the vector form of the Jacobi method.
2. Given again the initial vector $\mathbf{x}^{(0)} = \begin{pmatrix}1\\1\\1\end{pmatrix}$, use now the **vector form** of the Jacobi method to obtain $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$.

Solution

For $A = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 4 \end{bmatrix}$ we have $D = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 4 \end{bmatrix}$, $L = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ and $U = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ Hence

$$T = \overbrace{\begin{bmatrix} \frac{1}{4} & 0 & 0 \\ 0 & \frac{1}{8} & 0 \\ 0 & 0 & \frac{1}{4} \end{bmatrix}}^{D^{-1}} \overbrace{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}}^{L+U} = \begin{bmatrix} 0 & \frac{1}{4} & 0 \\ \frac{1}{8} & 0 & \frac{1}{8} \\ 0 & \frac{1}{4} & 0 \end{bmatrix}$$

In addition: $\mathbf{c} = D^{-1}\mathbf{b} = \begin{bmatrix} 12 \\ \frac{3}{2} \\ 6 \end{bmatrix}$ If $\mathbf{x}^{(0)} = \begin{pmatrix}1\\1\\1\end{pmatrix}$, then

$$\mathbf{x}^{(1)} = T\mathbf{x}^{(0)} + \mathbf{c} = \begin{bmatrix} 0 & \frac{1}{4} & 0 \\ \frac{1}{8} & 0 & \frac{1}{8} \\ 0 & \frac{1}{4} & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 12 \\ \frac{3}{2} \\ 6 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{bmatrix} + \begin{bmatrix} 12 \\ \frac{3}{2} \\ 6 \end{bmatrix} = \begin{bmatrix} \frac{49}{4} \\ \frac{7}{4} \\ \frac{25}{4} \end{bmatrix}$$

as before.
Note that we do not need to compute L and U but $R = L + U$:

```python
import numpy as np

A = np.array([[4, -1, 0],[-1, 8, -1],[0, -1,4]], dtype=float)
b = np.array([48, 12, 24], dtype=float)
x0 = np.array([1, 1, 1], dtype=float)
D = np.diag(A)
R = A - np.diagflat(D)
x = (b - np.dot(R,x0)) / D
print(x)
```

```
[12.25  1.75  6.25]
```

3.2 Gauss-Seidel method

Split up the sum into parts with $j < i$ and $j > i$

$$E_i: \quad x_i = \frac{1}{a_{ii}}\left[\sum_{j=1}^{i-1} -a_{ij}x_j + \sum_{j=i+1}^{n} -a_{ij}x_j\right] + \frac{b_i}{a_{ii}}$$

Algorithm 3.1 Jacobi Iteration

```
1: Input: $n \times n$ matrix $A$. $n \times 1$ vector $\mathbf{b}$, $N_{\max}$, TOL, $\mathbf{x^{(0)}}$.
2: procedure JACOBI($A, \mathbf{b}, N_{max}$, TOL)
3:     Set $n = 1$
4:     while $n \leq N_{max}$ do
5:         for $i = 1$ to $n$ do
6:             Set $x_i = \frac{1}{a_{ii}}\left[-\sum_{j=1,j\neq i} a_{ij}x_j^{(0)} + b_i\right]$
7:         end for
8:         if $||\mathbf{x} - \mathbf{x}^{(0)}|| <$ TOL then
9:             break
10:            Output $(\mathbf{x})$
11:        end if
12:        Set $n = n + 1$
13:        Set $\mathbf{x}^{(0)} = \mathbf{x}$.
14:    end while
15: end procedure
16: Output: $p$: approximation to the solution of $f(p) = 0$.
```

Add index k to all x_j, $j \leq i$ and $k-1$ to x_j $(j > 1)$.

Gauss-Seidel method: Given $x^{(0)} = \begin{bmatrix} x_1^{(0)} \\ \vdots \\ x_n^{(0)} \end{bmatrix}$, compute $x^{(k)} = \begin{bmatrix} x_1^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix}$, $(k = 1, 2, ...)$ by

$$x_i^{(k)} = \frac{1}{a_{ii}}\Big[\sum_{j=1}^{i-1} -a_{ij}x_j^{(k)} + \sum_{j=i+1}^{n} -a_{ij}x_j^{(k-1)}\Big] + \frac{b_i}{a_{ii}}, \qquad i = 1, \dots, n$$

1. Write the Gauss-Seidel method in vector form:\ $\mathbf{x}^{(k)} = T\mathbf{x}^{(k-1)} + \mathbf{c}$

In other words, what are T and $\mathbf{c}$? *Hint:* $A = (D - L) - U$

2. Consider again

$$A = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 4 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 48 \\ 12 \\ 24 \end{pmatrix}.$$

   Verify that the Gauss-Seidel method in this case can be written as

$$\mathbf{x}^{(k)} = \begin{bmatrix} 0 & \frac{1}{4} & 0 \\ 0 & \frac{1}{32} & \frac{1}{8} \\ 0 & \frac{1}{128} & \frac{1}{32} \end{bmatrix} \mathbf{x}^{(k-1)} + \begin{pmatrix} 12 \\ 3 \\ \frac{27}{4} \end{pmatrix}$$

3. Obtain $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$, given $\mathbf{x}^{(0)} = [0; 0; 0]$. How good is the Gauss–Seidel method?} (Recall that $\mathbf{x} = [13; 4; 7]$.)

Note that $A\mathbf{x} = \mathbf{b}$ can be written as $

$$(D - L)\mathbf{x} = U\mathbf{x} + \mathbf{b}$$

So Gauss-Seidel iteration:

$$(D - L)\mathbf{x}^{(k)} = U\mathbf{x}^{(k-1)} + \mathbf{b}$$

Given, $\mathbf{x}^{(0)} \in \mathbb{R}^n$, compute $\mathbf{x}^{(k)} = T\mathbf{x}^{(k-1)} + \mathbf{c}$ where

$$T = (D - L)^{-1}U$$
$$\mathbf{c} = (D - L)^{-1}\mathbf{b}$$

assuming that $(D - L)^{-1}$ exists. For part 2: Note that

$$D - L = \begin{bmatrix} 4 & 0 & \\ -1 & 8 & 0 \\ 0 & -1 & 4 \end{bmatrix}, \qquad U = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \qquad T = \begin{bmatrix} 0 & \frac{1}{4} & 0 \\ 0 & \frac{1}{32} & \frac{1}{8} \\ 0 & \frac{1}{128} & \frac{1}{32} \end{bmatrix}$$

and

$$\mathbf{c} = \begin{pmatrix} 12 \\ 3 \\ \frac{27}{4} \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 48 \\ 12 \\ 24 \end{pmatrix}$$

part 3:

$$\mathbf{x}^{(1)} = T \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \mathbf{c} = \begin{pmatrix} 12 \\ 3 \\ \frac{27}{4} \end{pmatrix}$$

$$\mathbf{x}^{(1)} = T\mathbf{x}^{(1)} + \mathbf{c} = T \begin{pmatrix} 12 \\ 3 \\ \frac{27}{4} \end{pmatrix} + \mathbf{c} = \begin{pmatrix} 12 + \frac{3}{4} \\ 3 + \frac{15}{16} \\ 6 + \frac{63}{64} \end{pmatrix}$$

Why are iterative techniques useful?

The Jacobi and Gauss-Seidel method require, per iteration step $\mathcal{O}(n^2)$ operations (for doing $T\mathbf{x}^{(k-1)} + \mathbf{c}$). If m steps are needed to obtain a sufficiently accurate approximation $\mathbf{x}^{(m)}$, then the cost is $m\mathcal{O}(n^2)$. Gaussian elimination requires $\mathcal{O}(n^3)$. Iterative methods are much more efficient when $m \ll n$ (few iterations and very large systems).

3.3 Convergence of iterative methods

Our aim now is to show that Jacobi's and Gauss-Seidel's method converges. However, for the proof we need some fundamental theory of convergent matrices.

3.3.1 Convergent matrices

Definition 3.4. A **matrix norm** on the set of all $n \times n$ matrices is a real-valued function $\|\cdot\|$, defined on this set, satisfying for all $n \times n$ matrices A and B satisfies the following properties:

1. $\|A\| \geq 0$
2. $\|A\| = 0$ if any only if A is O, the matrix with all zero entries.
3. $\|A\| = \|\alpha\| \, \|A\|$ for any scalar $\alpha \in \mathbb{R}$
4. $\|A + B\| \leq \|A\| + \|B\|$
5. $\|AB\| \leq \|A\| \, \|B\|$

Theorem 3.3. *If $\|\cdot\|$ is a vector norm on $\mathbb{R}^n$, then*

$$\|A\| = \max_{\|\boldsymbol{x}=1\|} \|A\boldsymbol{x}\|$$

is a matrix norm.

Note

One important property of a matrix norm is that:

$$\|A\mathbf{z}\| \leq \|A\| \, \|\mathbf{z}\| \qquad \forall \mathbf{z} \in \mathbb{R}^n$$

Indeed: let $\mathbf{z} \neq 0$, then

$$\|A\mathbf{z}\| = \frac{\|A\mathbf{z}\|}{\|\mathbf{z}\|} \|\mathbf{z}\| \leq \left(\max_{\mathbf{y} \neq 0} \frac{\|A\mathbf{y}\|}{\|\mathbf{y}\|} \right) \|\mathbf{z}\|$$

$$\left(\max_{\mathbf{y} \neq 0} \left\| A \frac{\mathbf{y}}{\|\mathbf{y}\|} \right\| \right) \|\mathbf{z}\| = \underbrace{\left(\max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\| \right)}_{=\|A\|} \|\mathbf{z}\|$$

Exercise 3.5. Consider the matrix $A = \begin{bmatrix} 3 & 5 \\ 4 & 0 \end{bmatrix}$

1. Compute $\|A\|_\infty = \max_{\|\mathbf{x}\|_\infty = 1} \|A\mathbf{x}\|_\infty$
2. Compute $\|A\|_2 = \max_{\|\mathbf{x}\|_2 = 1} \|A\mathbf{x}\|_2$

Solution

$$A\mathbf{x} = \begin{bmatrix} 3 & 5 \\ 4 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3x_1 + 5x_2 \\ 4x_1 \end{bmatrix}$$

Thus

$$\|A\|_\infty = \max_{\|\mathbf{x}\|_\infty = 1} \|A\mathbf{x}\|_\infty = \max_{\|\mathbf{x}\|_\infty = 1} \left\| \begin{bmatrix} 3x_1 + 5x_2 \\ 4x_1 \end{bmatrix} \right\|_\infty$$

When $\|\mathbf{x}\|_\infty = 1$ means that $\max\{|x_1|, |x_2|\} = 1$, Say for example $|x_1| = 1$ then $x_1 = \pm 1$ and $-1 \leq x_2 \leq 1$. Suppose that $x_1 = 1$, then

$$-2 \leq 3x_1 + 5x_2 = 3 + 5x_2 \leq 8$$

Hence the maximum of $|3x_{1}+5x_|$ is attained for $x_1 = 1$ and $x_2 = 1$. In this case:

$$\|A\|_\infty = \max_{\|\mathbf{x}\|_\infty = 1} \left\| \begin{bmatrix} 3x_1 + 5x_2 \\ 4x_1 \end{bmatrix} \right\|_\infty = \max\{8, 4\} = 8$$

If $x_1 = -1$, then

$$-8 \leq 3x_1 + 5x_2 = -3 + 5x_2 \leq 2$$

and so the maximum of $|3x_1 + 5x_|$ is attained for $x_1 = -1$ and $x_2 = -1$. In this case, we also have

$$\|A\|_\infty = \max_{\|\mathbf{x}\|_\infty = 1} \left\| \begin{bmatrix} 3x_1 + 5x_2 \\ 4x_1 \end{bmatrix} \right\|_\infty = \max\{8, 4\} = 8$$

For the l_2 norm:

$$\|A\|_2 = \max_{\|\mathbf{x}\|_2 = 1} \|A\mathbf{x}\|_2 = \max_{\|\mathbf{x}\|_2 = 1} \left\| \begin{bmatrix} 3x_1 + 5x_2 \\ 4x_1 \end{bmatrix} \right\|_2$$

and

$$\begin{aligned} \max_{\|\mathbf{x}\|_2 = 1} \left\| \begin{bmatrix} 3x_1 + 5x_2 \\ 4x_1 \end{bmatrix} \right\|_2 &= \max_{\|\mathbf{x}\|_2 = 1} \sqrt{(3x_1 + 5x_2)^2 + (4x_1)^2} \\ &= \max_{\|\mathbf{x}\|_2 = 1} \sqrt{25(x_1^2 + x_2^2) + 30x_1x_2)} \\ &= \max_{\|\mathbf{x}\|_2 = 1} \sqrt{25 + 30x_1x_2)} = \sqrt{40} \approx 6.32 \end{aligned}$$

Or you can simply:

```
A = np.array([[3, 5],[4,0]], dtype=float)
print(np.linalg.norm(A, ord=2))
print(np.linalg.norm(A, ord=np.inf))
```

```
6.324555320336759
8.0
```

Theorem 3.4. *If $A = (a_{ij})$ is an $n \times n$ matrix, then*

$$\|A\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |a_{ij}|$$

Definition 3.5. The *spectral radius* of a matrix A is defined by

$$\rho(A) = \max |\lambda|$$

where λ is an eigenvalue of A. (For complex $\lambda = \alpha + i\beta$, we define $|\lambda| = (\alpha^2 + \beta^2)^{1/2}$)

Definition 3.6. If A is an $n \times n$ matrix, then

1. $\|A\|_2 = \left[\rho(A^tA)\right]^{1/2}$
2. $\rho(A) \leq \|A\|$ for any natural norm $\|\cdot\|$

Exercise 3.6. Consider again the matrix $A = \begin{bmatrix} 3 & 5 \\ 4 & 0 \end{bmatrix}$

1. Compute $\|A\|_2$
2. Compute $\rho(A)$
3. Is $\rho(A) \leq \|A\|_\infty$ and $\rho(A) \leq \|A\|_2$?

Solution

Note that

$$A^tA = \begin{bmatrix} 3 & 4 \\ 5 & 0 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 4 & 0 \end{bmatrix} = \begin{bmatrix} 25 & 15 \\ 15 & 25 \end{bmatrix}$$

To compute eigenvalues we need to solve

$$\det \begin{bmatrix} 25-\lambda & 15 \\ 15 & 25-\lambda \end{bmatrix} = (25-\lambda)^2 - 15^2 = 0$$

which gives $\lambda = 10$ and $\lambda = 40$ Therefore,

$$\|A\|_2 = \left[\rho(A^tA)\right]^{1/2} = \left[\max\{10, 40\}\right]^{1/2} = \sqrt{40} \approx 6.32$$

2. We need

$$\det \begin{bmatrix} 3-\lambda & 5 \\ 4 & -\lambda \end{bmatrix} = (3-\lambda)(-\lambda) - 20 = \lambda^2 - 3\lambda - 20 = 0$$

$$\lambda = \frac{3}{2} \pm \frac{1}{2}\sqrt{89}$$

Thus

$$\rho(A) = \max|\lambda| = \frac{3}{2} + \frac{1}{2}\sqrt{89} \approx 6.21$$

3. Yes, $\rho(A) \leq \|A\|_\infty = 8$ (from previous exercise) and $\rho(A) \leq \|A\|_2 \approx 6.32$

Definition 3.7. An $n \times n$ matrix A is said to be convergent if

$$\lim_{k\to\infty} (A^k)_{i,j} = 0, \qquad \text{for each } i = 1\ldots, n, \quad \text{and } j = 1\ldots, n$$

Theorem 3.5. *The following statements are equivalent*

1. *A is a convergent matrix*
2. $\lim_{n\to\infty} \|A^n\| = 0$*, for some natural norm.*
3. $\lim_{n\to\infty} \|A^n\| = 0$*, for all natural norms.*
4. $\rho(A) < 1$
5. $\lim_{n\to\infty} A^n\boldsymbol{x} = \boldsymbol{0}$*, for every* $\boldsymbol{x}$

Exercise 3.7. Show that $A = \begin{bmatrix} 0 & \frac{1}{2} \\ \frac{1}{4} & 0 \end{bmatrix}$ is a convergent matrix

Solution

$$A^2 = \begin{bmatrix} 0 & \frac{1}{2} \\ \frac{1}{4} & 0 \end{bmatrix} \begin{bmatrix} 0 & \frac{1}{2} \\ \frac{1}{4} & 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{8} & 0 \\ 0 & \frac{1}{8} \end{bmatrix} = \frac{1}{8}I$$

So,

$$A^4 = A^2A^2 = \frac{1}{8^2}I^2$$

and

$$A^3 = AA^2 = A\frac{1}{8}I = \frac{1}{8}A$$

Then,

$$A^5 = A^2A^3 = \frac{1}{8}I\frac{1}{8}A = \frac{1}{8^2}A.$$

In general we have

$$A^{2k} = \frac{1}{8^k}I$$
$$A^{2k+1} = \frac{1}{8^k}A$$

for $k = 1, 2, \ldots$. Note that $A^k \to O$ as $k \to \infty$ since $A^{2k} \to O$ and $A^{2k+1} \to O$. Thus A is convergent.

Exercise 3.8. For matrix of Exercise 3.7 verify Theorem 3.5.

Solution

1. He have shown that A is convergent
2. $$\|A^{2k}\|_\infty = \left\|\frac{1}{8^k}I\right\|_\infty = \frac{1}{8^k}\|I\|_\infty = \frac{1}{8^k} \to 0, \quad as\, k \to \infty$$

   and

   $$\|A^{2k+1}\|_\infty = \left\|\frac{1}{8^k}A\right\|_\infty = \frac{1}{8^k}\|A\|_\infty = \frac{1}{8^k}\frac{1}{2} \to 0, \quad as\, k \to \infty$$

   since

   $$\|A\|_\infty = \left\|\begin{bmatrix} 0 & \frac{1}{2} \\ \frac{1}{4} & 0 \end{bmatrix}\right\|_\infty = \max\left\{\frac{1}{2}, \frac{1}{4}\right\} = \frac{1}{2}$$

3. Note that the eigenvalues of A are given by

$$\det\begin{bmatrix} -\lambda & \frac{1}{2} \\ \frac{1}{4} & -\lambda \end{bmatrix} = \lambda^2 - \frac{1}{8} \implies \lambda = \pm\frac{1}{\sqrt{8}}$$

So $\rho(A) = \frac{1}{\sqrt{8}} < 1$.

5. Note that

   $$A^{2k}\mathbf{x} = \frac{1}{8^k}I\mathbf{x} = \frac{1}{8^k}\mathbf{x} = \frac{1}{8^k}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \to \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

   as $k \to \infty$. Similarly,

   $$A^{2k+1}\mathbf{x} = \frac{1}{8^k}A\mathbf{x} = \frac{1}{8^k}\begin{bmatrix} 0 & \frac{1}{2} \\ \frac{1}{4} & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \frac{1}{8^k}\begin{bmatrix} \frac{x_2}{2} \\ \frac{x_1}{4} \end{bmatrix} \to \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

   as $k \to \infty$. Therefore, $\lim_{n\to\infty} A^n\mathbf{x} = \mathbf{0}$.

3.3.2 Convergence of Jacobi and Gauss-Seidel methods

Theorem 3.6. *For any $\boldsymbol{x}^0 \in \mathbb{R}^n$, the sequence $\{\boldsymbol{x}^{(k)}\}_{k=0}^\infty$ defined by*

$$\boldsymbol{x}^{(k)} = T\boldsymbol{x}^{k-1} + \boldsymbol{c}, \qquad \textit{for each } k \geq 1$$

converges to the unique solution

$$\boldsymbol{x} = T\boldsymbol{x} + \boldsymbol{c} \tag{3.6}$$

if and only if $\rho(T) < 1$.

Lemma 3.1. *If the spectral radius satisfies $\rho(T) < 1$, then $(I - T)^{-1}$ exists and*

$$(I - T)^{-1} = I + T + T^2 + \cdots = \sum_{j=0}^{\infty} T^j$$

Note

The previous lemma is a generalisation of the scalar power series

$$\frac{1}{1-\alpha}=\sum_{j=0}^{\infty}\alpha^j$$

which we know is valid provided $|\alpha|<1$.

Exercise 3.9. Consider the matrix $A=\begin{bmatrix}0 & \frac{1}{2}\\ \frac{1}{4} & 0\end{bmatrix}$

1. Compute $[I-A]^{-1}$.
2. Compute $I+A+A^2+A^3+\cdots$ Hint: Recall that $A^{2k}=\frac{1}{8^k}I$ and $A^{2k+1}=\frac{1}{8^k}A$,, for $k=1,2,\dots$.
3. For what matrix A do the above computations give distinct answers?

Solution

1.

$$\sum_{j=0}^{\infty}A^j=I+A+\sum_{j=1}^{\infty}[A^{2j}+A^{2j+1}]=I+A+\sum_{j=1}^{\infty}\frac{1}{8^k}[I+A]$$

so

$$\begin{aligned}\sum_{j=0}^{\infty}A^j&=\begin{bmatrix}1&0\\0&1\end{bmatrix}+\begin{bmatrix}0&\frac{1}{2}\\ \frac{1}{4}&0\end{bmatrix}+\sum_{j=1}^{\infty}\frac{1}{8^k}\left[\begin{bmatrix}1&0\\0&1\end{bmatrix}+\begin{bmatrix}0&\frac{1}{2}\\ \frac{1}{4}&0\end{bmatrix}\right]\\&=\begin{bmatrix}1&\frac{1}{2}\\ \frac{1}{4}&1\end{bmatrix}+\sum_{j=1}^{\infty}\frac{1}{8^k}\begin{bmatrix}1&\frac{1}{2}\\ \frac{1}{4}&1\end{bmatrix}\\&=\begin{bmatrix}1+\sum_{j=1}^{\infty}\frac{1}{8^k} & \frac{1}{2}\left[1+\sum_{j=1}^{\infty}\frac{1}{8^k}\right]\\ \frac{1}{4}\left[1+\sum_{j=1}^{\infty}\frac{1}{8^k}\right] & 1+\sum_{j=1}^{\infty}\frac{1}{8^k}\end{bmatrix}\end{aligned}$$

using the fact that

$$\sum_{j=0}^{\infty}\left(\frac{1}{8}\right)^j=\frac{1}{1-\frac{1}{8}}=\frac{8}{7}$$

we have

$$\sum_{j=0}^{\infty}A^j=\frac{8}{7}\begin{bmatrix}1&\frac{1}{2}\\ \frac{1}{4}&1\end{bmatrix}$$

On the other hand

$$[I-A]^{-1}=\begin{bmatrix}1&-\frac{1}{2}\\-\frac{1}{4}&1\end{bmatrix}^{-1}=\frac{8}{7}\begin{bmatrix}1&\frac{1}{2}\\ \frac{1}{4}&1\end{bmatrix}$$

which is the same as 1.

Proof of Theorem 3.6. ($\rho(T)<1$ implies convergence) Assume $\rho(T)<1$. Consider $\mathbf{x}^{(k)}$ given by

$$\mathbf{x}^{(k)}=T\mathbf{x}^{k-1}+\mathbf{c},\qquad \text{for each}\quad k\geq 1$$

Apply recursion

$$\begin{aligned}\mathbf{x}^{(k)}&=T\mathbf{x}^{k-1}+\mathbf{c}\\&=T\left(T\mathbf{x}^{k-2}+\mathbf{c}\right)+\mathbf{c}=T^2\mathbf{x}^{k-2}+[T+I]\mathbf{c}\\&=T^2\left[T\mathbf{x}^{k-3}+\mathbf{c}\right]+[T+I]\mathbf{c}=T^3\mathbf{x}^{k-3}+[T^2+T+I]\mathbf{c}\\&\vdots\\&=T^k\mathbf{x}^{(0)}+\left[T^{k-1}+T^{k-2}+\cdots+T+I\right]\mathbf{c}\end{aligned}$$

Since $\rho(T) < 1$ we have from Theorem X that $T^k\mathbf{x}^{(0)} \to 0$ as $k \to \infty$. Thus when we have the limit from both sides of the previous expression we have

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = \cancel{\lim_{k\to\infty} T^k\mathbf{x}^{(0)}} 0 + \sum_{k=1}^{\infty} T^j\mathbf{c}$$

and by using Lemma 3.1 we have

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = \cancel{\lim_{k\to\infty} T^k\mathbf{x}^{(0)}} 0 + [I-T]^{-1}\mathbf{c}$$

Thus, $\mathbf{x}^{(k)}$ has a limit $\hat{\mathbf{x}} \equiv \lim_{k\to\infty} \mathbf{x}^{(k)}$ and the limit is given by $\hat{\mathbf{x}} = [I-T]^{-1}\mathbf{c}$. Note that

$$[I-T]\hat{\mathbf{x}} = \mathbf{c} \Longrightarrow \hat{\mathbf{x}} = T\hat{\mathbf{x}} + \mathbf{c}$$

so the limit satisfies Equation 3.6 which completes the proof.

Let us now assume convergence, i.e. $\mathbf{x}^{(k)} = T\mathbf{x}^{(k-1)} + \mathbf{c}$ converges to $\mathbf{x}$ as $k \to \infty$ for any $\mathbf{x}^{(0)} \in \mathbb{R}^n$. From Theorem 3.5 it is sufficient to prove that $\lim_{k\to\infty} T^k\mathbf{z} = \mathbf{0}$ for all $\mathbf{z} \in \mathbb{R}^n$.

Let us choose $x^{(0)} = \mathbf{x} - \mathbf{x}^{(k)}$ and note that

$$\begin{aligned} \mathbf{x} - \mathbf{x}^{(k)} &= T\mathbf{x} + \mathbf{c} - \left[T\mathbf{x}^{(k-1)} + \mathbf{c}\right] \\ & T(\mathbf{x} - \mathbf{x}^{(k)}) \end{aligned}$$

so we can apply recursion

$$\begin{aligned} \mathbf{x} - \mathbf{x}^{(k)} &= T(\mathbf{x} - \mathbf{x}^{(k-1)}) \\ & T^2(\mathbf{x} - \mathbf{x}^{(k-2)}) \\ & \vdots T^k(\mathbf{x} - \mathbf{x}^{(0)}) = T^k\mathbf{z} \end{aligned}$$

Taking the limit on both sides

$$\lim_{k\to\infty} \left[\mathbf{x} - \mathbf{x}^{(k)}\right] = \lim_{k\to\infty} T^k\mathbf{z}$$

and since by assumption $\lim_{k\to\infty} \left[\mathbf{x} - \mathbf{x}^{(k)}\right] \to 0$ we have that $\lim_{k\to\infty} T^k\mathbf{z} = 0$. Since $\mathbf{z}$ was arbitrary, then $\lim_{k\to\infty} T^k\mathbf{z} = 0$ for all $\mathbf{z} \in \mathbb{R}^n$. Again from Theorem 3.5 we have that $\rho(T) < 1$. □

Corollary 3.2.

$$\left\|\boldsymbol{x} - \boldsymbol{x}^{(k)}\right\| \leq \|T\|^k \left\|\boldsymbol{x} - \boldsymbol{x}^{(0)}\right\|$$

–>

Chapter 4

Polynomial interpolation

4.1 Introduction to interpolation

- Suppose we have the value of a function of a single real variable, $f(x)$, at some points $\hat{x}_0, \dots, \hat{x}_n$ and we want to construct an approximation to the function at other points.
- One approach is to find a polynomial that agrees with values of f at $\hat{x}_0, \dots, \hat{x}_n$.
- This is called polynomial interpolation.
- Why do this?
  - f may be very expensive to evaluate e.g. it may be the output from a computer program that takes a long time to run
  - f may not be known at other values e.g. the known values may be measurements from an experiment.
  - Polynomial approximations are often building blocks for other numerical methods.

4.1.1 The linear interpolant

Given a function f and points $\hat{x}_0$ and $\hat{x}_1$, we wish to find the linear interpolant $p_1(x)$.

Approach 1: Let $p_1(x) = a + bx$.

We know that:

$$\begin{aligned} f(x_0) &= a + b\hat{x}_0 \\ f(x_1) &= a + b\hat{x}_1 \end{aligned}$$

Hence

$$\begin{bmatrix} 1 & \hat{x}_0 \\ 1 & \hat{x}_1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} f(\hat{x}_0) \\ f(\hat{x}_1) \end{bmatrix} \Longrightarrow \begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\hat{x}_1 - \hat{x}_0} \begin{bmatrix} \hat{x}_0 & -\hat{x}_1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} f(\hat{x}_0) \\ f(\hat{x}_1) \end{bmatrix}$$

which, of course, requires the two points $\hat{x}_0$ and $\hat{x}_1$ to be distinct.

Note

Approach 1 require the solution of a system of equations.

Approach 2: The aim is to write

$$p_1(x) = g(x) + h(x)$$

This can be done if we define

$$g(x) = f(\hat{x}_0)L_0(x) \qquad \text{and} \qquad h(x) = f(\hat{x}_1)L_1(\hat{x}_1)$$

So we can write $p_1(x)$ as

$$p_1(x) = f(\hat{x}_0)L_0(x) + f(\hat{x}_1)L_1(x).$$

with

$$L_0(x) = \frac{x-\hat{x}_1}{\hat{x}_0-\hat{x}_1}, \qquad L_1(x) = \frac{x-\hat{x}_0}{\hat{x}_1-\hat{x}_0}$$

Note that

$$\begin{aligned} L_0(\hat{x}_0) &= 1, & L_0(\hat{x}_1) &= 0, \\ L_1(\hat{x}_1) &= 1, & L_1(\hat{x}_0) &= 0 \end{aligned}$$

so $p_1(\hat{x}_0) = f(\hat{x}_0)$ and $p_1(\hat{x}_1) = f(\hat{x}_1)$.

Exercise 4.1. Let p_1 be the linear interpolant of the function $f(x) = x^4$ with interpolation points $\hat{x}_0 = -1$ and $\hat{x}_1 = 2$.

- Determine $p_1(x)$.
- Sketch $f(x)$ and $p_1(x)$.
- Compute the error at $x = 1$, $x = 2$ and $x = 3$.

Solution

We use Approach 2:

$$\begin{aligned} L_0(x) &= \frac{x-\hat{x}_1}{\hat{x}_0-\hat{x}_1} = \frac{x-2}{-1-2} = \frac{2-x}{3} \\ L_1(x) &= \frac{x-\hat{x}_0}{\hat{x}_1-\hat{x}_0} = \frac{x+1}{3} \end{aligned}$$

Furthermore, $f(\hat{x}_0) = (-1)^4 = 1$ and $f(\hat{x}_1) = 2^4 = 16$. Therefore,

$$p_1(x) = f(\hat{x}_0)L_0(x) + f(\hat{x}_1)L_1(x) = \frac{2-x}{3} + 16\left[\frac{x+1}{3}\right] = 5x+6$$

We define the error

$$\mathcal{E}(x) = |f(x) - p_1(x)|$$

We have that

$$\begin{aligned} \mathcal{E}(1) &= |f(1) - p_1(1)| = |1-11| = 10 \\ \mathcal{E}(2) &= |f(2) - p_1(2)| = 0 \\ \mathcal{E}(3) &= |f(3) - p_1(3)| = |81-21| = 60 \end{aligned}$$

4.1.2 The general problem and Lagrange's approach

Suppose a function $f(x)$ is evaluated at some points $\underbrace{\hat{x}_0, \hat{x}_1 \dots, \hat{x}_n}_{n+1 \text{ points}}$ (distinct).

We wish to construct the interpolating polynomial $p_n(x)$ of degree $\leq n$, so that

$$p_n(\hat{x}_j) = f(\hat{x}_j), \qquad \forall\ j = 0, 1, \dots, n$$

Approach 1. Let

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n = \sum_{j=0}^{n} a_j x^j.$$

We need to find $\{a_j\}_{j=0}^n$ for which $p_n(\hat{x}_j) = f(\hat{x}_j)$, for all $j = 0, \dots, n$. This leads to the matrix problem:

$$\overbrace{\begin{bmatrix} 1 & \hat{x}_0 & \hat{x}_0^2 & \dots & \hat{x}_0^n \\ 1 & \hat{x}_1 & \hat{x}_1^2 & \dots & \hat{x}_1^n \\ 1 & \hat{x}_2 & \hat{x}_2^2 & \dots & \hat{x}_2^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \hat{x}_n & \hat{x}_n^2 & \dots & \hat{x}_n^n \end{bmatrix}}^{V} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f(\hat{x}_0) \\ f(\hat{x}_1) \\ f(\hat{x}_2) \\ \vdots \\ f(\hat{x}_n) \end{bmatrix}$$

The $(n+1)\times(n+1)$ matrix V is called Vandermonde matrix. The cost of solving this system is $\mathcal{O}(n^3)$. In addition, V is ill-conditioned. Thus round-off error pollutes the solution as n is increased.Therefore, Approaxh 1 is not recommended.

Approach 2. Let $\mathbb{P}_n$ denote the set of polynomials of degree $\leq n$. We can write $\mathbb{P}_n$ in terms of a basis. For example, in Approach 1 the basis that we use is $\{1, x, x^2, \dots, x^n\}$ and so

$$\mathbb{P}_n = \text{span}\{1, x, x^2, \dots, x^n\}.$$

For approach the idea is to find a better basis to work with.

Definition 4.1. Let $L_j(x)$ be the polynomial of degree n such that

$$L_j(\hat{x}_j) = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \qquad i, j = 0, 1, 2, \dots, n \tag{4.1}$$

$L_0(x), L_1(x), \dots L_n(x)$ are called Lagrange polynomials

The set of Lagrange polynomials are a basis for $\mathbb{P}_n$, i.e.

$$\mathbb{P}_n = \text{span}\{L_0(x), L_1(x), \dots L_n(x)\}$$

and thus

$$p_n(x) = f(\hat{x}_0)L_0(x) + f(\hat{x}_1)L_1(x) + \dots + f(\hat{x}_n)L_n(x) = \sum_{j=0}^{n} f(\hat{x}_j)L_j(x) \tag{4.2}$$

Note

From Equation 4.1 we have that

$$p_n(\hat{x}_i) = \sum_{j=0}^{n} f(\hat{x}_j)L_j(x_i) = \sum_{j=0}^{n} f(\hat{x}_j)\delta_{ij} = f(\hat{x}_i)$$

Lagrange polynomials are given by the following expression

$$L_j(x) = \frac{\prod\limits_{\substack{i=0 \\ i\neq j}}^{n} (x - \hat{x}_i)}{\prod\limits_{\substack{i=0 \\ i\neq j}}^{n} (\hat{x}_i - \hat{x}_j)}$$

Exercise 4.2. Assume $n = 2$ (quadratic polynomials).

Let $x_0 = 1$, $x_1 = -1$ and $x_2 = 2$.

- Determine $L_0(x)$.
- Draw a picture of $L_0(x)$.
- Draw a picture of $L_1(x)$ and $L_2(x)$.

Solution

$$L_0(x) = \frac{(x - \hat{x}_1)(x - \hat{x}_2)}{(\hat{x}_0 - \hat{x}_1)(\hat{x}_0 - \hat{x}_2)} = \frac{(x + 1)(x - 2)}{(1 + 1)(1 - 2)} = \frac{2 + x - x^2}{2}$$

Exercise 4.3. Find the interpolating polynomial for a function f, given that $f(x) = 0, -3$, and 4 when $x = 1$, -1 and 2, respectively.

Solution

We have

$$\hat{x}_0 = 1, \quad \hat{x}_1 = -1, \quad \hat{x}_2 = 2$$

In addition

$$p_2(x) = \cancelto{0}{f(1)}L_0(x) + f(-1)L_1(x) + f(2)L_2(x)$$

Now

$$L_1(x) = \frac{(x-\hat{x}_0)(x-\hat{x}_2)}{(\hat{x}_1-\hat{x}_0)(\hat{x}_1-\hat{x}_2)} = \frac{(x-1)(x-2)}{(-1-1)(-1-2)} = \frac{(x-1)(x-2)}{6}$$

$$L_2(x) = \frac{(x-\hat{x}_0)(x-\hat{x}_1)}{(\hat{x}_2-\hat{x}_0)(\hat{x}_2-\hat{x}_1)} = \cdots = \frac{x^2-1}{3}$$

Hence

$$p_2(x) = -3\frac{(x-1)(x-2)}{6} + 4\frac{x^2-1}{3} = \frac{1}{6}(5x^2+9x-14)$$

We can verify that $p_2(1) = 0$, $p_2(-1) = -3$ and $p_2(2) = -4$.

Exercise 4.4 (Uniqueness of polynomial interpolant)**.** Let p_2 be the interpolating polynomial of degree at most 2 that interpolates f at the distinct points $\hat{x}_0$, $\hat{x}_1$, and $\hat{x}_2$. Consider $n = 2$ (quadratic interpolant).

Assuming that $\hat{x}_0$, $\hat{x}_1$, and $\hat{x}_2$ are distinct prove that p_2 is unique.

Solution

Assume there is another interpolating polynomial $q_2(x)$. Thus

$$p_2(\hat{x}_j) = f(\hat{x}_j), \qquad j = 0,1,2$$
$$q_2(\hat{x}_j) = f(\hat{x}_j), \qquad j = 0,1,2$$

Let $v_2(x) = p_2(x) - q_2(x)$ so $v_2(x)$ is a polynomial of degree at most 2 and

$$v_2(\hat{x}_j) = 0 \qquad j = 0,1,2$$

i.e. $v_2(x)$ has three roots: $\hat{x}_0$, $\hat{x}_1$ and $\hat{x}_2$.
From the Fundamental Theorem of Algebra we know that any nonzero polynomial of degree n has at most n real roots.
Since $v_2(x)$ has 3 roots v_2 has to be the zero polynomial, i.e.

$$v_2(x) \equiv 0 \implies p_2(x) \equiv q_2(x)$$

Hence $p_2(x)$ is unique.

Note

The argument above holds for all polynomial of degree n. Also note that existence of p_n is guaranteed because we have a way of constructing it.

4.2 Error Analysis

Please review Section A.3 for the notation on function spaces.

Theorem 4.1. *Suppose $\hat{x}_0, \dots, \hat{x}_n$ are distinct numbers in the interval $[a,b]$ and $f \in C^{n+1}[a,b]$. Let p_n the interpolant defined in (Equation 4.2). Then, for each $x \in [a,b]$, there exist a number (generally unknown) $\xi(x) \in (a,b)$ such that*

$$f(x) = p_n(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-\hat{x}_0)(x-\hat{x}_1)\cdots(x-\hat{x}_n)$$

Proof. Consider the case $n = 1$.

Subcase 1.1: Suppose $x^* = \hat{x}_0$ or $x^* = \hat{x}_1$. Then, $f(x^*) = p_1(x^*)$ so $f(x^*) - p_1(x^*) = 0$.

Also

$$\frac{f''(\xi(x^*))}{2!}(x^* - \hat{x}_0)(x^* - \hat{x}_1) = 0$$

so the results hold.

Subcase 1.2: Let $x^* \neq \hat{x}_0$ and $x^* \neq \hat{x}_1$. Define:

$$q(x) = f(x) - p_1(x) - \lambda^* g(x) \tag{4.3}$$

where $\lambda \in \mathbb{R}$ (is yet to be defined) and where

$$g(x) = (x - \hat{x}_0)(x - \hat{x}_1).$$

Note that $g(x^*) \neq 0$ because $x^* \neq x_0, x_1$. Also, $q(\hat{x}_0) = 0$ and $q(\hat{x}_1) = 0$.

Let us now set λ^* so that $q(x^*) = 0$, that is

$$q(x^*) = f(x^*) - p_1(x^*) - \lambda^* g(x^*) = 0 \Longrightarrow \lambda^* = \frac{f(x^*) - p_1(x^*)}{g(x^*)}.$$

With this choice of λ^* have that $q(x) = 0$ at $x \in \{\hat{x}_0, \hat{x}_1, x^*\}$. Therefore, since we also have that $q \in C^2[a,b]$, we can apply Generalised Rolle's (Theorem A.5) to ensure the existence of $\xi^* \in [a,b]$ such that $q''(\xi^*) = 0$.

Now:

$$q''(x) = f''(x) - \cancelto{0}{p_1''(x)} - \lambda^* \cancelto{2}{g''(x)} = f''(x) - 2\lambda^*$$

since $p_1(x)$ is at most linear. Therefore,

$$q''(\xi^*) = 0 \Longrightarrow f''(\xi^*) - 2\lambda^* = 0 \Longrightarrow \lambda^* = \frac{f''(\xi^*)}{2}$$

using this expression for λ^* in (Equation 4.3) evaluated at $x = x^*$ gives us

$$0 = q(x^*) = f(x^*) - p_1(x^*) - \lambda^* g(x^*) = f(x^*) - p_1(x^*) - \frac{f''(\xi^*)}{2} g(x^*)$$

and using the definition of $g(x)$ we have

$$f(x^*) = p_1(x^*) + \frac{f''(\xi^*)}{2}(x^* - \hat{x}_0)(x^* - \hat{x}_1).$$

For the general nth case we can define

$$g(t) = f(t) - p_n(t) - \lambda(t - x_0)(t - x_1) \cdots (t - x_n)$$

and follow the same steps □

4.2.1 Application I: How good is the interpolant?

Exercise 4.5 (Error bound for interpolation)**.** To approximate a smooth function f for $x \in [x_0, x_1]$, consider the linear polynomial $p_1(x)$ that interpolates f at x_0 and x_1.

1. Compute a bound on the maximum (absolute) error.
2. What is this bound in case of $f(x) = \exp(x)$, $x_0 = 0$, $x_1 = 1$?

Solution

For 1. we note from Theorem 4.1 that

$$f(x) = p_1(x) + \frac{f''(\xi)}{2}(x - \hat{x}_0)(x - \hat{x}_1)$$

for $\xi \in (x_0, x_1)$. Thus

$$\begin{aligned}|f(x) - p_1(x)| &= \frac{|f''(\xi)|}{2}\big|(x-\hat{x}_0)(x-\hat{x}_1)\big| \\ &\le \frac{1}{2}\max_{\xi\in[x_0,x_1]}|f''(\xi)|\max_{\xi\in[x_0,x_1]}\big|(x-\hat{x}_0)(x-\hat{x}_1)\big|\end{aligned}$$

Note that $(x-\hat{x}_0)(x-\hat{x}_1)$ has a minimum (which is negative) at $\zeta = \frac{\hat{x}_0+\hat{x}_1}{2}$, so $\big|(x-\hat{x}_0)(x-\hat{x}_1)\big|$ has a maximum at $\zeta = \frac{\hat{x}_0+\hat{x}_1}{2}$.
Also note that

$$\big|(\zeta-\hat{x}_0)(\zeta-\hat{x}_1)\big| = \left[\frac{\hat{x}_1-\hat{x}_0}{2}\right]^2.$$

Hence,

$$\begin{aligned}|f(x) - p_1(x)| &\le \frac{1}{2}\overbrace{\max_{\xi\in[x_0,x_1]}|f''(\xi)|}^{:=M_2}\max_{\xi\in[x_0,x_1]}\big|(x-\hat{x}_0)(x-\hat{x}_1)\big| \\ &= \frac{M_2}{8}(\hat{x}_1-\hat{x}_0)^2\end{aligned}$$

where we have defined

$$M_2 = \max_{\xi\in[x_0,x_1]}|f''(\xi)|$$

which exists since $f \in C^2[a,b]$.
For part 2. we note that since $f(x) = e^x$ then

$$M_2 = \max_{\xi\in[x_0,x_1]}\Big\{|f''(\xi)|\Big\} = \max_{\xi\in[x_0,x_1]} e^x = e$$

Thus,

$$|f(x) - p_1(x)| \le \frac{M_2}{8}(\hat{x}_1-\hat{x}_0)^2 = \frac{e}{8}(1-0) = \frac{e}{8}$$

4.2.2 Piecewise polynomial interpolation

- Piecewise polynomial interpolation is performed by splitting an interval $[a,b]$ into subintervals $[x_{i-1}, x_i]$, $i = 0, \dots, N$ so that

  $$a = x_0 < x_1 < x_2 < \cdots < x_N = b$$

  Then, the usual polynomial interpolant on each of these subintervals is used.
- Below is an example of piecewise linear inerpolation with $N = 3$, The piecewise interpolant is denoted by $S_1^3(x)$.

Piecewise linear interpolation is to be used to approximate $f(x) = e^{-2x}$ for $x \in [0,1]$. All subintervals $[x_{i-1}, x_i]$ have fixed length h.

1. What is the largest value of h needed to ensure that piecewise linear interpolation given an (absolute) error of at most 0.02.

2. What is the largest value of h needed to ensure that piecewise linear interpolation given an (absolute) error of at most 0.0002.

Solution

Note that $f''(x) = 4e^{-2x}$ thus $\max_{\xi\in[0,1]}|f''(\xi)| = 4$.
Let $S_1^N(x)$ denote the piecewise linear interpolant. Let us look at the nth interval$[x_{n-1}, x_n]$ on which $S_1^N(x)$ is the linear interpolant.
Following the same steps that we employed for the solution to Problem **??** we have that, on the subinterval $x \in [x_{n-1}, x_n]$, the following holds:

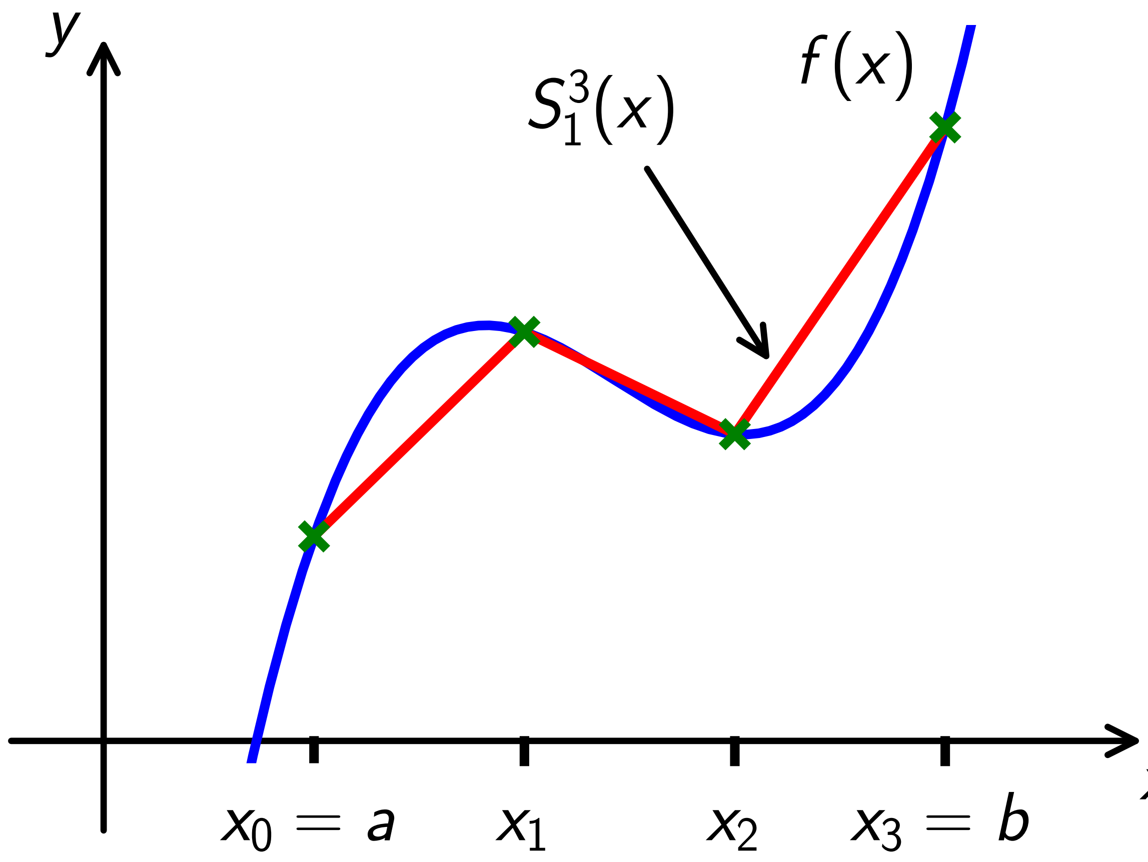


Figure 4.1: piecewise polynomial interpolation

$$\begin{aligned}|f(x) - S_1^N(x)| &\le \frac{1}{2} \max_{\xi\in[x_{n-1},x_n]} |f''(\xi)| \max_{\xi\in[x_{n-1},x_n]} \big|(x-\hat{x}_{n-1})(x-\hat{x}_n)\big| \\ &= \frac{1}{8} \max_{\xi\in[x_{n-1},x_n]} |f''(\xi)| (\underbrace{\hat{x}_n - \hat{x}_{n-1}}_{=h})^2 = \frac{h^2}{8} \max_{\xi\in[x_{n-1},x_n]} |f''(\xi)|\end{aligned}$$

for $n = 1, 2, \dots, N$.
Note that

$$\max_{\xi\in[x_{n-1},x_n]} |f''(\xi)| \le \max_{x\in[0,1]} |f''(\xi)| \le 4$$

Hence

$$|f(x) - S_1^N(x)| \le \frac{h^2}{8} \overbrace{\max_{\xi\in[x_{n-1},x_n]} |f''(\xi)|}^{=4} \le \frac{4h^2}{8} = \frac{h^2}{2}, \qquad x \in [x_{n-1}, x_n]$$

Since the nth interval was chosen arbitrarily, we have that the maximum absolute error over the entire interval satisfies

$$\max_{x\in[0,1]} |f(x) - S_1^N(x)| \le \frac{h^2}{2} \tag{4.4}$$

Thus if we want the error to be less than 0.02:

$$\max_{x\in[0,1]} |f(x) - S_1^N(x)| \le \frac{h^2}{2} \le 0.02 \implies h \le \sqrt{2(0.02)} = 0.2$$

To answer 2. we see from Equation 4.4 that the error is proportional to h^2, in other the error is $\mathcal{O}(h^2)$ (see next section). Hence decreasing h by a factor of 10 means that the error is decreased by a factor of 100. Thus with $h = 0.02$ the error is 0.0002.

4.2.3 Big $\mathcal{O}$ Notation

Definition 4.2. Given two functions $f, g : \mathbb{R} \to \mathbb{R}$, we write $f(h) = \mathcal{O}(g(h))$ as $h \to 0$ if there exists $M > 0$ and $\delta > 0$ such that

$$|f(h)| \leq C|g(h)| \qquad \text{if} \quad |h| < \delta$$

We say f is Big $\mathcal{O}$ of g as $h \to 0$.

Example 4.1. $h + h^2 = \mathcal{O}(h^2)$ as $h \to 0$. Indeed, since $h \to 0$ we have that $|h| < \delta = 1$ and so

$$|h + h^2| = |h||h+1| \leq |h|(|h|+1) \leq 2|h|$$

Hence $h + h^2 = \mathcal{O}(h^2)$ for $M = 2$ and $\delta = 1$.

Example 4.2. $e^x = \mathcal{O}(1)$. From Taylor's expansion we have that

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

and so $e^x = \mathcal{O}(1)$ as $x \to 0$. We can also write this as

$$\begin{aligned} e^x &= \mathcal{O}(1), \qquad \text{as } x \to 0 \\ e^x &= 1 + \mathcal{O}(x), \qquad \text{as } x \to 0 \\ e^x &= 1 + x + \mathcal{O}(x^2), \qquad \text{as } x \to 0 \\ &\vdots \end{aligned}$$

* Further examples: + $f(h) = 2h^2 + 3h^3 + h^4 = \mathcal{O}(h^2)$ as $h \to 0$ (the low order dominates). + $f(h) = 1 + 4h^3 = \mathcal{O}(1)$ as $h \to 0$. + $\sin(h) = \mathcal{O}(h)$ (recall $\sin(h) \approx h$ for small h) * This Big $\mathcal{O}$ notation is commonly used when discussing the errors of a numerical method.

> **Note**
>
> Suppose $f(h) = \mathcal{O}(h^n)$, then $f(h) \approx Mh^n$ for small enough h. then
>
> $$\log|f(h)| = \log(M|h^n|) = \log M + n \log|h|$$

Theorem 4.2. *Suppose that $a = x_0 < x_1 < x_2 < \cdots < x_N = b$ are a set of uniformly separated points, with distance h between them. Let $S_p^n(x)$ be the piecewise pth order polynomial interpolant of $f(x) \in C^{p+1}[a,b]$, then*

$$\max_{x\in[a,b]} |f(x) - S_p^n(x)| \leq Ch^{p+1} \max_{x\in[a,b]} |f^{(p+1)}(x)|$$

for some constant C, independent of h.

In other words, the error is $\mathcal{O}(h^{p+1})$ as $h \to 0$.

Proof. Note that

$$\max_{x\in[a,b]} |f(x) - S_p^n(x)| = \max_{1\leq k\leq n} \max_{x\in[x_{k-1},x_k]} |f(x) - p_p^k(x)|$$

where p_p^k is the pth-degree interpolant on the interval $[x_{k-1}, x_k]$ with nodes $\hat{x}_0^k, \hat{x}_1^k, \ldots, \hat{x}_p^k$.

By Using Theorem 4.1 and the fact that if $x \in [x_{k-1}, x_k]$ then $|x - \hat{x}_j^k| \leq h$ we have

$$\begin{aligned} \max_{x\in[a,b]} |f(x) - S_p^n(x)| &= \max_{1\leq k\leq n} \max_{x\in[x_{k-1},x_k]} |f(x) - p_p^k(x)| \\ &\leq \max_{1\leq k\leq n} \max_{x\in[x_{k-1},x_k]} \left| \frac{f^{(p+1)}(\xi_k(x))}{(p+1)!} \right| \overbrace{|x - \hat{x}_0^k|}^{\leq h}, \overbrace{|x - \hat{x}_1^k|}^{\leq h} \cdots \overbrace{|x - \hat{x}_p^k|}^{\leq h} \\ &\leq \max_{x\in[a,b]} |f^{(p+1)}(\xi(x))| \frac{h^{p+1}}{(p+1)!} \end{aligned}$$

where $\xi(x) = \xi_k(x)$ for some k (hence $\xi \in [a,b]$). Let $C = \frac{1}{(p+1)!}$ and the results follows. $\square$

- Each line on the plot below is for a fixed polynomial degree p and making the mesh width h smaller, for $f(x) = \sin(x)$,
- We see $\mathcal{O}(h^p + 1)$ convergence as we expect.
- Note that for $p = 6$ and small h we observe the effect of round-off error.

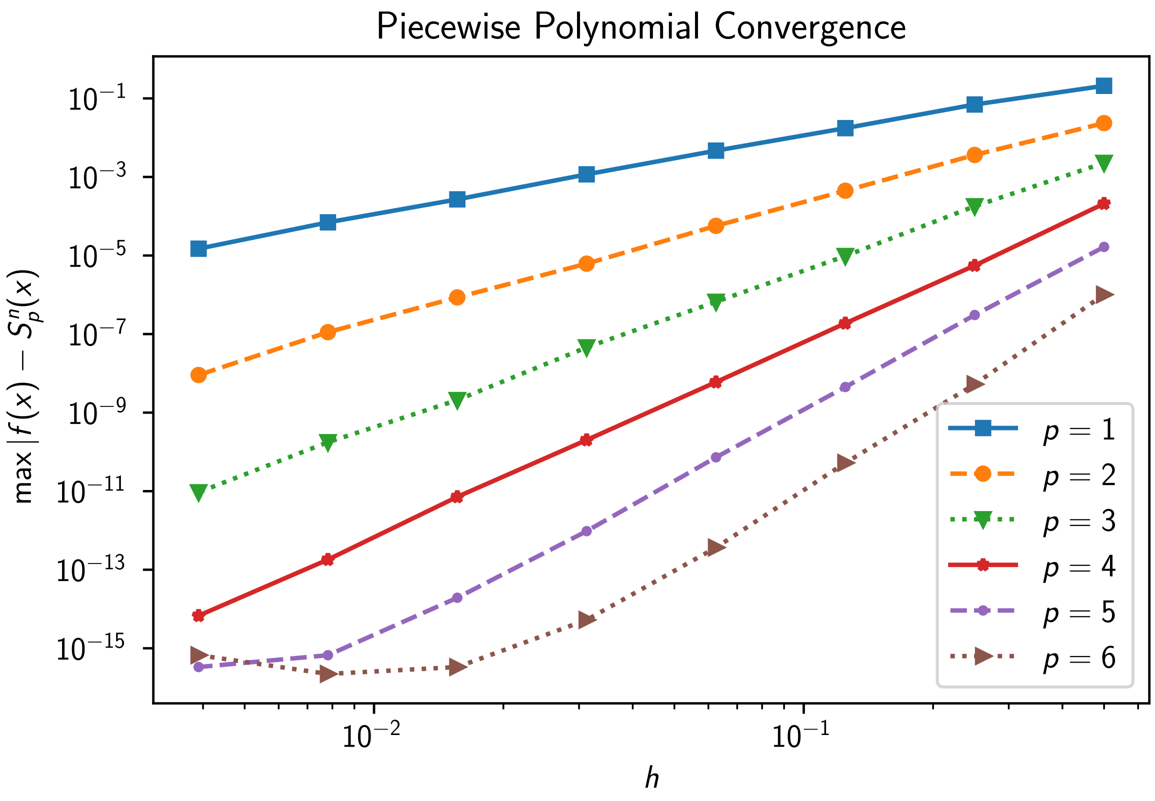


Figure 4.2: polynomial comvergence

Chapter 5

Numerical Calculus

5.1 Numerical Differentiation

5.1.1 Introduction

Given $f(x_0)$ and $f(x_0 + h)$, we wish to approximate $f'(x_0)$.

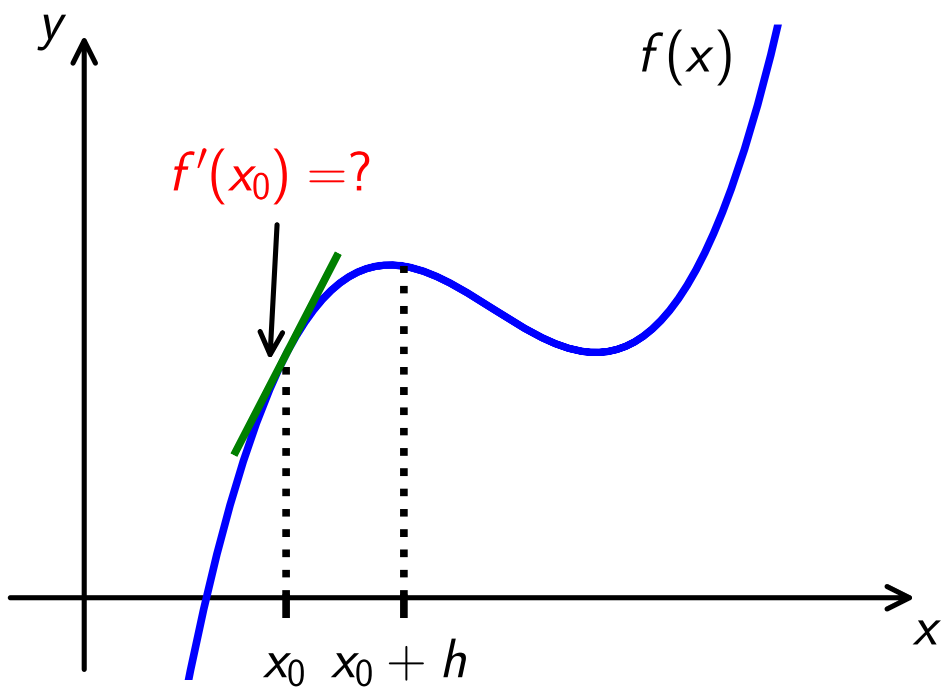


Figure 5.1: Approximating $f'(x_0)$

Recall

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

Definition 5.1.

1. Forward difference quotient:
$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h} \tag{5.1}$$
2. Backward difference quotient:
$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h} \tag{5.2}$$

Exercise 5.1. Let $f(x) = x - x^2 - x^3$. We are interested in approximating $f'(0) = 1$.

1. Compute the value of the forward-difference quotient with step h.
2. Compute the value of the backward-difference quotient with step h.
3. Compute the value of the central-difference quotient with step h.
4. Find the errors for the above two approximations, and their asymptotic behaviour as $h \to 0$.

Solution

1. $x_0 = 0$:
$$f'(0) \approx \frac{f(h) - f(0)}{h} = \frac{h - h^2 - h^3 - 0}{h} = 1 - h - h^2$$
2.
$$f'(0) \approx \frac{f(0) - f(-h)}{h} = \frac{0 - (h - h^2 + h^3)}{h} = 1 + h - h^2$$
3. Error: Note that $f'(x) = 1 - 2x - 3x^2$ and so $f'(0) = 1$. Thus
$$\text{Error} = f'(0) - \text{approx} = \begin{cases} h + h^2 & \text{for } 1 \\ -h + h^2 & \text{for } 2 \end{cases}$$

As $h \to 0$, the asymptotic order of convergence is $\mathcal{O}(h)$ in both cases.

Exercise 5.2. Consider the difference approximation from Equation 5.1.

1. Derive the error representation
$$f'(x_0) - \left[\frac{f(x_0 + h) - f(x_0)}{h}\right] = -\frac{h}{2} f''(\xi) \tag{5.3}$$
for some ξ between x_0 and $x_0 + h$.
2. Derive a bound on the absolute value of the error 3. Verify the bound for the examples in Exercise 5.1.

Solution

Par 1.: From Taylor theorem (Theorem A.3)
$$f(x_0 + h) = f(x_0) + h f'(x_0) + \frac{h^2}{2} f''(\xi)$$
for $\xi \in (x_0, x_0 + h)$. Solving for $f'(x_0)$ we have
$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \frac{1}{h}\frac{h^2}{2} f''(\xi)$$
which gives the result in Equation 5.3.
Alternative proof of part 1.: Let
$$p_1(x) = f(x_0) L_0(x) + f(x_1) L_1(x)$$
where $x_1 = x_0 + h$ and
$$L_0(x) = \frac{x - x_1}{x_0 - x_1}, \qquad L_1(x) = \frac{x - x_0}{x_1 - x_0}, \tag{5.4}$$
so $p_1(x)$ is the usual linear interpolant.
From (**err-pol?**) we have
$$f(x) = p_1(x) + \frac{f''(\xi(x))}{2}(x - x_0)(x - \overset{x_0+h}{\overbrace{x_1}}).$$
Differentiating we have
$$f'(x) = p_1'(x) + \frac{f''(\xi(x))}{2}(2(x - x_0) - h) + \left[\frac{d}{dx}\frac{f''(\xi(x))}{2}\right](x - x_0)(x - x_1)$$
when $x = x_0$ we have
$$f'(x_0) = p_1'(x_0) + \frac{f''(\xi(x))}{2}(2(x_0 - x_0) - h) + \left[\frac{d}{dx}\frac{f''(\xi(x))}{2}\right]_{x_0} \cancelto{0}{(x_0 - x_0)}(x_0 - x_1)$$

so

$$f'(x_0) = p_1'(x_0) + \frac{f''(\xi(x))}{2}(-h) \tag{5.5}$$

note that

$$p_1'(x) = f(x_0)L_0'(x) + f(x_1)L_1'(x).$$

From Equation 5.4 we see that

$$L_0'(x) = \frac{1}{x_0 - x_1} = -\frac{1}{h}, \qquad L_1'(x) = \frac{1}{x_1 - x_0}\frac{1}{h},$$

and so

$$p_1'(x_0) = \frac{f(x_0+h) - f(x_0)}{h}$$

Combining Equation 5.5 with the previous expression we have

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} + \frac{f''(\xi(x))}{2}(-h)$$

which shows Equation 5.3.
For part 2. we note that

$$\left| f'(x_0) - \left[\frac{f(x_0+h) - f(x_0)}{h}\right]\right| = \frac{h}{2}|f''(\xi)| \leq \frac{h}{2} \max_{\xi\in[x_0,x_0+h]} |f''(\xi)| = \mathcal{O}(h)$$

Verify the bound for $f(x) = x - x^2 - x^3$. Note that $f'(x) = 1 - 2x - 3x^2$ and $f''(x) = -2 - 6x = -2(1+3x)$. Then,

$$\left|f'(0) - \frac{f(h) - f(0)}{h}\right| \leq \max_{\xi\in[0,h]} |2 + 6\xi|\frac{h}{2} = (2+6h)\frac{h}{2} = h + 3h^2$$

From Exercise 5.1 we have:

$$\left|f'(0) - \frac{f(h) - f(0)}{h}\right| = |h + h^2| = h + h^2 \leq h + 3h^2$$

which verifies the error bound.

Exercise 5.3 (Python Demo.)**.** Investigate a difference-formula approximation to the derivative of $\sin(x)$ at $x = 0.9$ for various values of h.

1. Can you observe the expected asymptotic behavious as $h \to 0$? For example, take $h = 0.1, 0.01, 0.001, \dots$ and compute the corresponding errors
2. What happens for very small h?

Solution

Let $\hat{f}(x+h)$ and $\hat{f}(x)$ denote the computer representation of $f(x+h)$ and $f(x)$ respectively. We have

$$\hat{f}(x+h) = f(x+h) + \epsilon_1$$
$$\hat{f}(x) = f(x) + \epsilon_2$$

where $\epsilon_1 \ll 1$ and $\epsilon_2 \ll 1$. Then

$$\frac{\hat{f}(x+h) - \hat{f}(x)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{\epsilon_1 - \epsilon_2 + \epsilon_3}{h} = \mathcal{O}(h) + \mathcal{O}(h^{-1})$$

5.1.2 $(n+1)$-point formulae

Problem: Given $f(x_0), f(x_1), \dots, f(x_n)$, approximate $f'(x_j)$ for $j = 1, \dots, n$.

Main idea: Use derivative of polynomial interpolant:

$$f'(x_j) \approx p_n'(x_j)$$

Example: $n = 2$ with points $\{x_j\}_{j=0}^2$ and let

$$f_j \equiv f(x_j) \qquad j = 0, 1, 2.$$

Assume uniformly spaced points with distance h between them. So $x_j = x_0 + jh$ with $h > 0$. Let $j = 0$ and approximate $f'(x_0)$. We have

$$f(x) \approx \sum_{i=0}^{2} f_i L_i(x)$$

where $L_i(x)$ are Lagrange polynomials. Then,

$$f'(x_0) \approx \sum_{i=0}^{2} f_i L_i'(x_0).$$

Recall

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{\underbrace{(x_0 - x_1)}_{-h}\underbrace{(x_0 - x_2)}_{-2h}}$$

then

$$L_0'(x) = \frac{1}{2h^2}(2x_0 - (x_1 + x_2)) = -\frac{3h}{2h^2} = -\frac{3}{2h}.$$

Similarly $L_1'(x) = \frac{4}{2h}$ and $L_2'(x_0) = -\frac{1}{2h}$. Hence:

$$f'(x_0) \approx \frac{1}{2h}(-3f_0 + 4f_1 - f_2).$$

Let $j = 1$, then

$$f'(x_1) \approx \sum_{i=0}^{2} f_i L_i'(x_1)$$

It us not difficult to see that $L_0'(x_1) = -\frac{1}{2h}$, $L_1'(x_1) = 0$ and $L_2'(x_1) = \frac{1}{2h}$. So

$$f'(x_1) \approx \frac{1}{2h}(f_2 - f_0)$$

the case $j = 2$ is left as an exercise.

Exercise 5.4. Let $f(x) = x - x^2 - x^3$. We are interested in approximating $f'(0) = 1$.

1. Compute the value of the central difference quotient with step h. Central difference quotient with step$h$ is defined by

$$f'(x_0 \approx \frac{f(x_0 + h) + f(x_0 - h)}{2h}$$

2. Find the errors for the above two approximations, and their asymptotic behaviour as $h \to 0$.

Solution

$j = 1$ $x_0 = -h$ and $x_1 = 0$ and $x_2 = h$

$$f'(0) \approx \frac{f(h) - f(-h)}{2h} = \frac{h - h^2 - h^3 - (-h - h^2 + h^3)}{2h} = 1 - h^2$$

on the other hand the exact value of $f'(0)$ is $f'(0) = 1$

$$\text{error} = f'(0) - \left[\frac{f(h) - f(-h)}{2h}\right] = 1 - (1 - h^2) = h^2 = \mathcal{O}(h^2)$$

which means the approximation is more accurate than forward/backward differences.

5.2 Numerical Integration

5.2.1 Introduction

The problem: Given $f_0 = f(x_0)$, $f_1 = f(x_1)$, approximate $I = \int_a^b f(x)\,dx$. The approximation:

$$I = \int_a^b f(x)\,dx \approx \frac{b-a}{2}(f_0 + f_1)$$

.

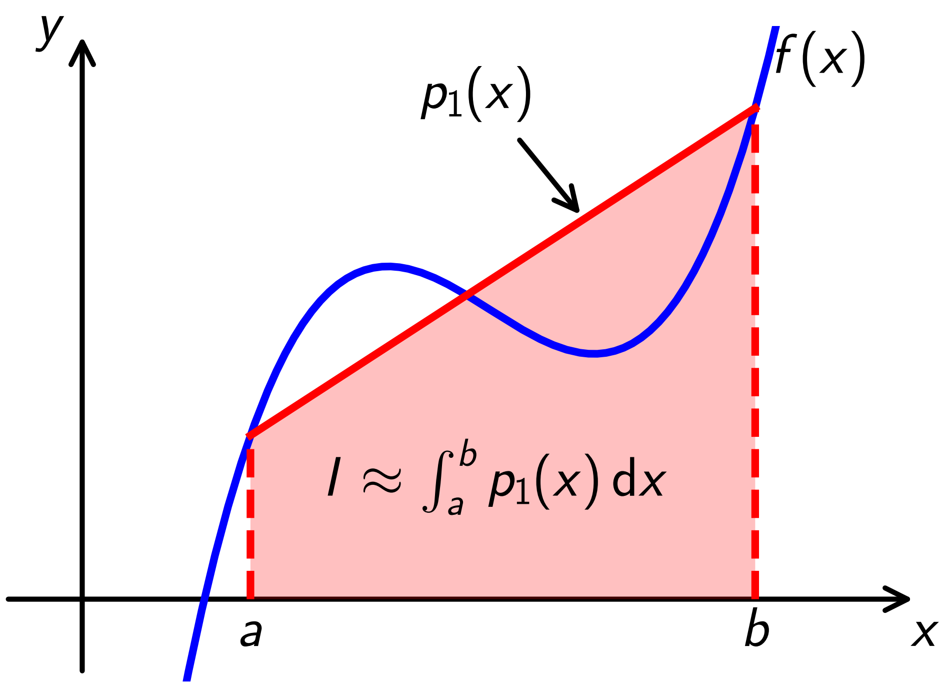


Figure 5.2: Approximating $\int_a^b f(x)dx$

Given $f_0 = f(x_0), f_1 = f(x_1), \dots, f_n = f(x_n)$, approximate $I = \int_a^b f(x)\,dx$.

Main idea: Integrate the polynomial interpolant:

$$I \approx \int_a^b p_n(x)\,dx$$

If $n = 1$: Trapezoidal/trapezium rule:

$$I = \frac{h}{2}(f_0 + f_1)$$

If $n = 2$: Simpson's rule:

$$I = \frac{h}{3}(f_0 + 4f_1 + f_2)$$

where $x_i - x_{i-1} = h$ for $i = 1, 2$.

Exercise 5.5. Approximate the integral

$$\int_0^1 (c_0 + c_1 x + c_2 x^2)\,dx = c_0 + \tfrac{1}{2}c_1 + \tfrac{1}{3}c_2$$

1. using the Trapezoidal rule ($h = 1$).
2. using Simpson's rule ($h = 1/2$).

How good are they?

Solution

1. Trapezoidal rule: $x_0 = 0$, $x_1 = 1$, $h = 1$.

$$I \approx \frac{1}{2}(f(0) + f(1)) = \frac{1}{2}(c_0 + c_0 + c_1 + c_2) = c_0 + \frac{c_1}{2} + \frac{c_2}{2}$$

Note that the absolute error is

$$\left| c_0 + \tfrac{1}{2}c_1 + \tfrac{1}{3}c_2 - \left(c_0 + \frac{c_1}{2} + \frac{c_2}{2}\right)\right| = \left|\frac{c_2}{3} - \frac{c_2}{2}\right| = \frac{c_2}{6}$$

2. Simpson's rule: x_0, $x_1 = \frac{1}{2}$, $x_2 = 1$, $h = \frac{1}{2}$.

$$\begin{aligned} I &\approx \frac{h}{3}(f(0) + 4f(1/2) + f(1)) = \frac{1}{6}\left[c_0 + 4\left(c_0 + \frac{c_1}{2} + \frac{c_2}{4}\right) + c_0 + c_1 + c_2\right] \\ &= c_0 + \frac{c_1}{2} + \frac{c_2}{3} \end{aligned}$$

Note that Simpson's rule is exact.

5.2.2 Integration rules with remainder

Theorem 5.1 (Trapezoidal rule with remainder)**.** *Suppose $h = (b - a)$, then (for sufficiently smooth f), we have*

$$I = \int_a^b f(x)dx = \frac{h}{2}(f_0 + f_1) - \frac{h^3}{12}f''(\xi)$$

for some $\xi \in (a, b)$.

Theorem 5.2 (Simpson's rule with remainder)**.** *Suppose $h = \frac{b-a}{2}$, then (for sufficiently smooth f), we have*

$$I = \int_a^b f(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2) - \frac{h^5}{90}f^{(4)}(\xi)$$

for some $\xi \in (a, b)$.

The **degree of accuracy or precision** of a quadrature formula is the largest positive integer n such that the formula is exact for x^k, for each $k = 0, 1, 2, \dots, n$.

We have Trapezoidal rule: $n = 1$ and Simposon's rule $n = 3$.

Exercise 5.6. What is the degree of accuracy of the midpoint rule

$$\int_a^b f(x)dx \approx hf\left(\frac{a+b}{2}\right) \qquad \text{with } h = b - a$$

Solution

Consider $f(x) = c_0$

$$\int_a^b f(x)dx = c_0(b - a), \qquad h\,f\left(\frac{a+b}{2}\right) = (b - a)c_0$$

Consider $f(x) = c_1x$

$$\int_a^b f(x)dx = \frac{c_1}{2}(b^2 - a^2), \qquad h\,f\left(\frac{a+b}{2}\right) = (b - a)c_1\left(\frac{a+b}{2}\right) = \frac{c_1}{2}(b^2 - a^2)$$

Consider $f(x) = c_2x^2$

$$\int_a^b f(x)dx = \frac{c_2}{3}(b^3 - a^3), \qquad h\,f\left(\frac{a+b}{2}\right) = (b - a)c_2\left(\frac{a+b}{2}\right)^2$$

Theorem 5.3 (Weighted Mean Value Theorem for Integrals)**.** *Suppose $f \in C[a,b]$, the Riemann integral of g existis on $[a,b]$, and $g(x)$ does not change sign on $[a,b]$. Then there exists a number c in (a,b) with*

$$\int_a^b f(x)g(x)dx = f(c)\int_a^b g(x)dx$$

Suppose $f \in C[a,b]$, then there exists a value c in (a,b) such that

$$\int_a^b f(x)dx = f(c)(b-a)$$

Alternatively:

$$f(c) = \frac{1}{b-a}\int_a^b f(x)dx$$

f attains its mean value.

Proof. Let $g(x) = 1 > 0$ then from Theorem 5.3 we have

$$\int_a^b f(x)\,1\,dx = f(c)\int_a^b 1\,dx = f(c)(b-a), \quad \text{where } c \in (a,b)$$

Hence

$$\frac{1}{b-a}\int_a^b f(x)dx = f(c), \quad \text{for some } c \in (a,b)$$

□

of Theorem Theorem 5.1. Use the error for $p_1(x)$ with $\hat{x}_0 = a$, $\hat{x}_1 = b$

$$f(x) = p_1(x) + \frac{f''(\xi(x))}{2}(x-a)(x-b)$$

for some $\xi(x) \in (a,b)$. Hence,

$$\begin{aligned}\int_a^b f(x)dx &= \int_a^b p_1(x)dx + \int_a^b \left[\frac{f''(\xi(x))}{2}\overbrace{(x-a)(x-b)}^{\le 0}\right]dx \\ &= \int_a^b p_1(x)dx + \frac{f''(\mu)}{2}\overbrace{\int_a^b \Big[(x-a)(x-b)\Big]dx}^{-\frac{h^3}{6}} \\ &= \frac{1}{2}[f(a)+f(b)] - \frac{h^3}{12}f''(\mu)\end{aligned}$$

for some $\nu \in (a,b)$. □

5.2.3 Composite Numerical Integration

Main idea: Approximate

$$I = \int_a^b f(x)dx$$

by dividing $[a,b]$ into subintervals/strips and applying a quadrature rule on each subinterval.

Suppose we have n strips and $n+1$ points, with $x_j = x_0 + jh$ where $h = \frac{b-a}{n}$, $j = 0,1,\ldots,n$. Composite Trapezoidal Rule (CTR)

$$\begin{aligned}I &= \sum_{j=1}^{n}\int_{x_{j-1}}^{x_j} f(x)dx \\ &\approx \sum_{j=1}^{n}\frac{h}{s}[f_{j-1}+f_j] = \frac{h}{2}\left[\sum_{j=1}^{n} f_{j-1} + \sum_{j=1}^{n} f_j\right] \\ &= \frac{h}{2}\Big[f_0 + 2\sum_{j=1}^{n} f_j + f_n\Big]\end{aligned}$$

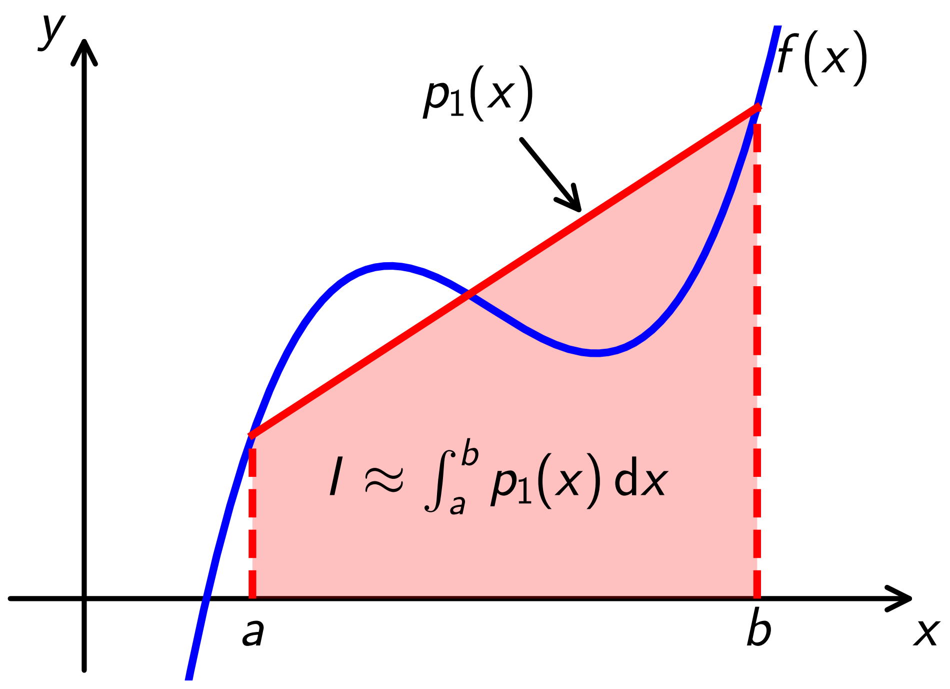


Figure 5.3: Composite numerical integration

Exercise 5.7. Approximate the integral $\int_0^1 \sin(\pi x)\, dx = \frac{2}{\pi} \approx 0.6366$

1. using the Composite Trapezoidal Rule (CTR) with 2 strips (= 3 points),
2. using the Composite Midpoint Rule (CMR) with 2 strips (= 2 points),
3. using the Composite Simpson's Rule (CSR) with 2 strips (= 5 points).

Solution

1. CTR:
$$\frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)]$$
where $h = \frac{1}{n}$, $x_j = jh$, $j = 0, 1, \ldots, n$.
2 strips, 3 points: $\frac{h}{2}$
$$\int_0^1 \sin(\pi x) dx \approx \frac{1}{4}[\sin(0) + 2\sin(\pi/2) + \sin(\pi)] = \frac{1}{2}$$
2. CMR:
$$\hat{h}[f(\hat{x}_0) + f(\hat{x}_1) + f(\hat{x}_2) + \cdots + f(\hat{x}_{n-1})]$$
where $\hat{h} = \frac{1}{n}$, $\hat{x}_j = \frac{\hat{h}}{2} + j\hat{h}$, $j = 0, 1, \ldots, n-1$.
2 strips, 2 points, $\hat{h} = \frac{1}{2}$
$$\int_0^1 \sin(\pi x) dx \approx \frac{1}{2}[\sin(\pi/4) + \sin(3\pi/4)] = \frac{1}{2}\left[\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}\right] \approx 0.7071$$
3. CSR:
$$\frac{\tilde{h}}{3}[f(\tilde{x}_0) + 4f(\tilde{x}_1) + 2f(\tilde{x}_2) + 4f(\tilde{x}_3) + \cdots + 2f(\tilde{x}_{2n-2}) + 4f(\tilde{x}_{2n-1}) + f(\tilde{x}_{2n})]$$
where $\tilde{h} = \frac{1}{2n}$, $\tilde{x}_j = j\tilde{h}$, $j = 0, 1, \ldots, 2n$.

2 strips, 5 points, $\tilde{h} = 1/4$.

$$\begin{aligned}\int_0^1 \sin(\pi x)dx &\approx \frac{1}{12}[\sin(0) + 4\sin(\pi/4) + 2\sin(\pi/2) + 4\sin(3\pi/4) + \sin(\pi)] \\ &= \frac{\sqrt{2}}{3} + \frac{1}{6} \approx 0.6381\end{aligned}$$

Theorem 5.4 (Compoisite Trapezoidal Rule with error term)**.** *Let $f \in C^2[a,b]$, $h = (b-a)/n$ and $x_j = a + jh$, for each $j = 0, 1, \dots, n$. There exists a $\mu \in (a,b)$ for which the Composite Trapezoidal rule for n subintervals can be written with its error term as*

$$\int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right] - \frac{b-a}{12}h^2 f''(\mu)$$

Solution

We have

$$\begin{aligned}\int_a^b f(x)dx = \sum_{j=1}^{n}\int_{x_{j-1}}^{x_j} f(x)dx &= \sum_{j=1}^{n}\left[\frac{h}{2}[f_{j-1} + f_j] - \frac{h^3}{12}f''(\mu_j)\right] \\ &= \underbrace{\frac{h}{2}[f_0 + \sum_{j=1}^{n} 2f_j + f_n]}_{CTR} - \frac{h^3}{12}n\sum_{j=1}^{n}\left[\frac{f''(\mu_j)}{n}\right]\end{aligned}$$

where $\mu_j \in (x_{j-1}, x_j)$ for all $j = 1, \dots, n$.
Note that

$$\min_{x\in[a,b]} f''(x) \le f''(\mu_j) \le \max_{x\in[a,b]} f''(x)$$

hence we also have

$$\overbrace{\frac{1}{n}\sum_{j=1}^{n}\min_{x\in[a,b]} f''(x)}^{\min_{x\in[a,b]} f''(x)} \le \frac{1}{n}\sum_{j=1}^{n} f''(\mu_j) \le \overbrace{\frac{1}{n}\sum_{j=1}^{n}\max_{x\in[a,b]} f''(x)}^{\max_{x\in[a,b]} f''(x)}.$$

By Theorem A.1, there exists $\mu \in (a,b)$ such that

$$f''(\mu) = \frac{1}{n}\sum_{j=1}^{n} f''(\mu_j) = \frac{1}{n}\sum_{j=1}^{n}\frac{f''(\mu_j)}{n}$$

Then,

$$\int_a^b f(x)dx = \frac{h}{2}[f_0 + \sum_{j=1}^{n} 2f_j + f_n] - \frac{h^3}{12}nf''(\mu)$$

and since $h = (b-a)/n$, then $n = (b-a)/h$. Thus,

$$\begin{aligned}\int_a^b f(x)dx &= \frac{h}{2}[f_0 + \sum_{j=1}^{n} 2f_j + f_n] - \frac{h^3}{12}\frac{b-a}{h}f''(\mu) \\ &= \frac{h}{2}[f_0 + \sum_{j=1}^{n} 2f_j + f_n] - \frac{h^2}{12}(b-a)f''(\mu)\end{aligned}$$

which proves the result.

Exercise 5.8 (How good is the composite rule approx.?)**.** Compute a bound on the (absolute value of the) error for the Composite Rule approximations in Exercise 5.7.

Solution

CTR:
$$\left|\int_a^b f(x)dx - CTR\right| \leq \frac{(b-a)}{12}h^2 \max_{\mu\in[a,b]}|f''(\mu)|. \tag{5.6}$$
For $f(x) = \sin(\pi x)$, $\max_{\mu\in[0,1]}|f''(\mu)| = \pi^2$. Also, $b-a=1$, $n=2$ so $h=(b-a)/n=1/2$. Hence:
$$|Error| \leq \frac{\pi^2}{48} \approx 0.206$$
CSR:
$$\left|\int_a^b f(x)dx - CSR\right| \leq \frac{(b-a)}{180}h^4 \max_{\mu\in[a,b]}|f^{(4)}(\mu)| \tag{5.7}$$
$h = \frac{b-a}{2n}$
CMR:
$$\left|\int_a^b f(x)dx - CMR\right| \leq \frac{(b-a)}{24}h^2 \max_{\mu\in[a,b]}|f''(\mu)| \tag{5.8}$$
$h = \frac{b-a}{n}$

Exercise 5.9 (How to guarantee a given accuracy?)**.** The integral $\int_0^1 \sin(\pi x)dx$ is to be approximated using CTR. How many strips are required to have an error less than 4×10^{-4}.

Solution

We know that
$$\left|\int_0^1 \sin(\pi x)dx - CTR\right| \leq \frac{h^2}{12}\max_{\mu\in[0,1]}\left|\pi^2\sin(\pi\mu)\right| = \frac{\pi^2h^2}{12} = \frac{\pi^2}{12n^2}$$
$h=\frac{1}{n}$. If $\frac{\pi^2}{12n^2} \leq 4\times 10^{-4}$, then so is the error
$$n \geq \sqrt{\frac{\pi^2}{(12)(4\times 10^{-4})}} \approx 45.3$$
so $n \geq 46$.

Exercise 5.10. The integral $\int_0^1 \sin(\pi x)dx$ is to be approximated using CTR. How many strips are required to have an error less than 1×10^{-4}.

- At least 12
- At least 23
- At least 33
- At least 46
- At least 65
- At least 91
- At least 182

Solution

We have that the error is $\mathcal{O}(h^2)$, and the desired errors has a reduced a factor of 1/4, so h has to be halve (or n has to double) so the answer is at least 182.

Exercise 5.11. The integral $\int_0^1 \sin(\pi x)dx$ is to be approximated using CMR. How many strips are required to have an error less than 1×10^{-4}.

- At least 10
- Approximately 20
- Approximately 45

- Approximately 65
- Approximately 130
- Approximately 180
- Way more than 180

Solution

From expression Equation 5.8 we have

$$\left|\int_a^b f(x)dx - CMR\right| \leq \frac{h^2\pi^2}{24}$$

Hence

$$\frac{h^2\pi^2}{24} = \frac{\pi^2}{24n^2} \leq 10^{-4} \implies n \geq \frac{\pi}{\sqrt{24 \times 10^{-4}}} \approx 64.127$$

thus $n \geq 65$.

Exercise 5.12. The integral $\int_0^1 \sin(\pi x)dx$ is to be approximated using CSR. How many strips are required to have an error less than 1×10^{-4}.

- At least 5
- Approximately 20
- Approximately 45
- Approximately 65
- Approximately 130
- Approximately 180
- Way more than 180

Solution

From Equation 5.7

$$\left|\int_a^b f(x)dx - CSR\right| \leq \frac{h^4\pi^4}{180}$$

with $h = \frac{1}{2n}$ Hence

$$\frac{h^4\pi^4}{180} = \frac{\pi^4}{(16)(180)n^4} \leq 10^{-4} \implies n \geq \left[\frac{\pi^4}{(16)(180)10^{-4}}\right]^{1/4} \approx 4.288$$

So $n \geq 5$ will be sufficient.

5.2.4 Richards Extrapolation

Motivating Example

Let $f(x) = x - x^2 - x^3$. Goal find $M = f'(0)$.

Approximation method: Use forward differences:

$$M \approx N_1(h) = \frac{f(h) - f(0)}{h}$$

Extrapolation Formula 1:

$$N_2 = 2N_1\left(\frac{h}{2}\right) - N_1(h)$$

Extrapolation Formula 2:

$$N_3 = \frac{4}{3}N_2\left(\frac{h}{2}\right) - \frac{1}{3}N_2(h)$$

Compute the values in a table:

$$\begin{tikzcd} N_1\left(\frac{1}{2}\right) \arrow[dr] & & \\ N_1\left(\frac{1}{4}\right) \arrow[r] & N_2\left(\frac{1}{2}\right) \arrow[dr] & \\ N_1\left(\frac{1}{8}\right) \arrow[r] & N_2\left(\frac{1}{4}\right) \arrow[r] & N_3\left(\frac{1}{2}\right) \end{tikzcd}$$

Exact answer: $M = f'(0) = 1$. \ First column: $N_1(h) = \frac{f(h)-f(0)}{h} = 1 - h + h^2$ Hence

$$N_1\left(\frac{1}{2}\right) = \frac{3}{4}$$
$$N_1\left(\frac{1}{4}\right) = \frac{13}{16}$$
$$N_1\left(\frac{1}{8}\right) = \frac{57}{64}$$

Second column:

$$N_2(h) = 2N_1\left(\frac{4}{2}\right) - N_1(h)$$

Hence

$$N_2\left(\frac{1}{2}\right) = 2N_1\left(\frac{1}{4}\right) - N_1\left(\frac{1}{2}\right) = \frac{7}{8}$$
$$N_2\left(\frac{1}{4}\right) = \dots = \frac{31}{32}$$

Third column:

$$N_3(h) = \frac{4}{3}N_2\left(\frac{h}{2}\right) - \frac{1}{3}N_2(h) = \left(\frac{4}{3}\right)\left(\frac{31}{32}\right) - \left(\frac{1}{3}\right)\left(\frac{7}{8}\right) = 1$$

5.2.5 Derivation of extrapolation formulae.

Suppose $f \in C^\infty[x_0, x_0 + h]$ then

$$M = f'(x_0) = \underbrace{\frac{f(x_0 + h) - f(x_0)}{h}}_{N_1(h)} + K_1 h + K_2 h^2 + K_3 h^3 + K_4 h^4 + \dots \tag{5.9}$$

Also

$$M = f'(x_0) = \underbrace{\frac{f(x_0 + h/2) - f(x_0)}{h/2}}_{N_1(h/2)} + K_1\left(\frac{h}{2}\right) + K_2\left(\frac{h^2}{4}\right) + K_3\left(\frac{h^3}{8}\right) + K_4\left(\frac{h^4}{16}\right) + \dots \tag{5.10}$$

Note that multiplying Equation 5.9 by 2 and subtracting Equation 5.10 gives

$$2M - M = 2N_1(h/2) - N_1(h) = \left(2K_1\frac{h}{2} - K_1 h\right) \underbrace{- \frac{K_2 h^2}{2} - \frac{3K_3 h^3}{4} + \dots}_{\mathcal{O}(h^2)}$$

We also have

$$M = N_2(h) - \frac{K_2 h^2}{2} - \frac{3K_3 h^3}{4} + \dots$$

and

$$M = N_2(h/2) - \frac{K_2 h^2}{8} - \frac{3K_3 h^3}{31} + \dots$$

We can find a and b such that

$$M = aN_2(h) + bN_2(h/2) + \mathcal{O}(h^3)$$

$$LHS : aM + bM = M \implies a + b = 1$$
$$RHS : -\frac{aK_2 h^2}{2} - \frac{-bK_2 h^2}{8} = (0)(h^2) \implies 4a + b = 0$$

Thus $a = -\frac{1}{3}$ and $b = \frac{4}{3}$ Hence

$$M = \overbrace{\frac{4}{3}N_2(4/2) - \frac{1}{3}N_2(h)}^{N_3(h)} + \mathcal{O}(h^3)$$

Exercise 5.13. Let $M = f'(0)$ and $N_1(h) = \frac{f(h)-f(0)}{h}$

1. Write $N_1(h/2)$ in terms of f.
2. Write $N_2(h)$ in terms of f
3. How good is $N_2(h)$? Pick for example $f(x) = x - x^2 - x^3$

Solution

1.

$$N_1(h/2) = \frac{2(f(h/2) - f(0))}{h}$$

2.

$$\begin{aligned} N_2(h) &= 2\frac{2(f(h/2) - f(0))}{h} - \frac{f(h) - f(0)}{h} \\ &= \frac{-3f(0) + 4f(h/2) - f(h)}{h} \end{aligned}$$

3. Error is $\mathcal{O}(h^2)$ for $N_2(h)$. If $f(x) = x - x^2 + x^3$, $f'(0) = 1$.

$$\begin{aligned} N_2(h) &= \frac{-3f(0) + 4f(h/2) - f(h)}{h} \\ &= 4\left(\frac{1}{2} - \frac{h}{4} + \frac{h^2}{8}\right) - (1 - h + h^2) \\ &= 1 - \frac{1}{2}h^2 \end{aligned}$$

Exercise 5.14. Denote by CTR(h) the CTR appoximation with interval width h to an integral $I = \int_a^b f(x)dx$. Assuming $f \in C^\infty[a, b]$, then it can be shown that

$$I = \text{CTR}(h) + \underbrace{K_1h^2 + K_2h^4 + K_3h^6 + ...}_{\mathcal{O}(h^2)}$$

Derive the Richardson extrapolation formula in this case.

K_1, K_2, K_3 ... depend only on derivatives of f at $x = a$ and $x = b$

Solution

Let $R_{1,1} = \text{CTR}(h_1)$, $R_{2,1} = \text{CTR}(h_2)$ with $h_2 = \frac{h_1}{2}$.

$$I = R_{1,1} + K_1h^2 + K_2h^4 + K_3h^6 + ... \tag{5.11}$$

and

$$I = R_{2,1} + K_1\left(\frac{h_1}{2}\right)^2 + K_2\left(\frac{h_1}{2}\right)^4 + K_3\left(\frac{h_1}{2}\right)^6 + ... \tag{5.12}$$

As before, find α and β such that

$$\alpha I + \beta I = I \implies \alpha + \beta = 1$$
$$\alpha K_1 h_1^2 + bK_1\frac{h_1^2}{4} = (0)(h^2) \implies 4\alpha + \beta = 0$$

Then $\alpha = -\frac{1}{3}$ and $\beta = \frac{4}{3}$. Then, α Equation 5.11+ β Equation 5.12 gives

$$I = \underbrace{\frac{4}{3}R_{2,1} - \frac{1}{3}R_{1,1}}_{R_{2,2}} + \mathcal{O}(h^4)$$

Exercise 5.15. Let $I = \int_a^b f(x)dx$. For $h_1 = b - a$ the CTR gives

$$R_{1,1} = \frac{h_1}{2}\Big(f(a) + f(b)\Big)$$

1. Write $R_{2,1}$ in terms of $f(a)$, $f\left(\frac{a+b}{2}\right)$, $f(b)$.
2. Write $R_{2,2}$ in terms of $f(a)$, $f\left(\frac{a+b}{2}\right)$, $f(b)$.
3. How good is $R_{2,2}$

Solution

1.
$$R_{2,1} = \frac{h_1}{4}\left(f(a) + 2f\left(\frac{a+b}{2}\right) + f(b)\right)$$

2. Note that:
$$\begin{aligned} R_{2,2} &= \frac{4}{3}R_{2,1} - \frac{1}{3}R_{1,1} \\ &= \frac{4}{3}\left[\frac{h_1}{4}\left(f(a) + 2f\left(\frac{a+b}{2}\right) + f(b)\right)\right] - \frac{1}{3}\left[\frac{h_1}{2}\Big(f(a) + f(b)\Big)\right] \\ &= \frac{h_1}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] \\ &= \frac{h_2}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] \end{aligned}$$

which is Simpson's rule with width h_2.

3. We know Simpson's rule has error $\mathcal{O}(h^4)$ which is consistent with extrapolation theory.

What formulae are obtained when we carry on ferforming Richardson Extrapolation with $R_{n,1} = \text{CTR}(h_n)$ and $h_n = \frac{h}{2^{n-1}}$?

5.2.6 General Romberg Integration

$$\begin{aligned} R_{n,2} &= \frac{4}{3}R_{n,1} - \frac{1}{3}R_{n-1,1}, && n \geq 2 \\ R_{n,3} &= \frac{16}{15}R_{n,2} - \frac{1}{15}R_{n-1,2}, && n \geq 3 \\ R_{n,4} &= \frac{64}{63}R_{n,3} - \frac{1}{63}R_{n-1,3}, && n \geq 4 \end{aligned}$$

Romberg Integration Rules on CTR:

Let $R_{n,1} = \text{CTR}(h_n)$. Then, for $\rho > 1$, we have

$$R_{n,p} = \frac{4^{(p-1)}}{4^{(p-1)} - 1}R_{n,p-1} - \frac{1}{4^{(p-1)} - 1}R_{n-1,p-1}, \qquad n \geq p$$

the RHS removes the $h^{2(p-1)}$ error term

Exercise 5.16. $2 = \int_0^\pi \sin(x)dx \approx CTR$.

Complete the following table:

Table 5.1: Romberg Integration

		1× extrap.	2×	3×
$h_1 = \pi$	$R_{1,1} =$			
$h_2 = \frac{h_1}{2}$	$R_{2,1} =$	$R_{2,2} =$		

		1× extrap.	2×	3×
$h_3 = \frac{h_1}{4}$	$R_{3,1} =$	$R_{3,2} =$	$R_{3,3} =$	
$h_4 = \frac{h_1}{8}$	$R_{4,1} =$	$R_{4,2} =$	$R_{4,3} =$	$R_{4,4} =$

Solution

Table 5.2: Romberg Integration

		1× extrap.	2×	3×
$h_1 = \pi$	$R_{1,1} = 0$			
$h_2 = \frac{h_1}{2}$	$R_{2,1} = 1.570$	$R_{2,2} = 2.0944$		
$h_3 = \frac{h_1}{4}$	$R_{3,1} = 1.896$	$R_{3,2} = 2.0046$	$R_{3,3} = 1.99857$	
$h_4 = \frac{h_1}{8}$	$R_{4,1} = 1.974$	$R_{4,2} = 2.0003$	$R_{4,3} = 1.99998$	$R_{4,4} = 2.000006$

Chapter 6

Numerical ODEs

6.1 Euler's Method

Problem:

Solve the Initial-Value Problem (IVP)

$$\begin{aligned}\frac{dy}{dt}(t) &= f(t, y(t)), \qquad a \leq t \leq b \\ y(a) &= y_0\end{aligned}$$

IVP= ODE +initial condition

Main idea:

- Divide $[a, b]$ into N intervals:
  - Step size: $h = \frac{b-a}{N}$
  - Nodes: $t_i = a + ih$, $i = 0, 1, \dots, N$
- Let $y_i \approx y(t_i)$, $i = 0, 1, \dots, N$
- Find y_i with the formula:

$$y_{i+1} = y_i + h\, f(t_i, y_i), \qquad i = 0, 1, \dots N - 1$$

- Of course, we already have y_0 from the IVP
- This is a difference equation with an initial condition

Derivation 1 - Taylor's Theorem

Consider Taylors's expansion of $y(t)$:

$$\begin{aligned}y(t_{i+1}) &= y(t_i + h) \\ &= y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \frac{h^3}{6}y'''(t_i) + \dots \\ &= y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2}y''(t_i) + \dots\end{aligned}$$

Let $y_i \approx y(t_i)$, $i = 0, 1, \dots, N - 1$ and truncate after the first two terms:

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)), \qquad i = 0, 1, \dots N - 1$$

note that $y_0 = y(0)$.

Derivation 2 -Quadrature rule

By the fundamental theorem of calculus we have

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} \frac{dy}{dt}\, dt = \int_{t_i}^{t_{i+1}} f(t, y(t))dt$$

Use quadrature rule to do the integration. In this case use the left rectangle rule:

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} \frac{dy}{dt} \approx h\, f(t_i, y(t_i))$$

Derivation 3 - Geometric

Recall that the solution $y(t)$ is a curve that is everywhere parallel to the vector $[1, f(t,y)]^t$ The (forward) Euler method approximates $y(t_{i+1})$ by moving along the vector $h[1, f(t,y)]^t$ from the position (t_i, y_i)

Exercise 6.1. Consider the IVP

$$\begin{aligned} y' &= y - t^2 + 1, \qquad 0 \le t \le 2 \\ y(0) &= \frac{1}{2} \end{aligned}$$

1. Use Euler's method with $h = \frac{1}{2}$ to obtain:

$$y_1 \approx y\left(\frac{1}{2}\right), \qquad y_2 \approx y(1), \qquad \dots$$

2. Compare with exact solution $y(t) = (t+1)^2 - \frac{1}{2}e^t$.

Solution

Euler's method:

$$y_{i+1} = y_i + h(y_i - t_i^2 + 1)$$

$t_i = \frac{i}{2}$, $i = 0, 1, 2, 3, 4$

$$\begin{aligned} y_0 &= \frac{1}{2} \\ y_1 &= \frac{1}{2} + \frac{1}{2}\left(\frac{1}{2} - 0 + 1\right) = \frac{5}{4} = 1.25 \\ y_2 &= \frac{5}{4} + \frac{1}{2}\left(\frac{5}{4} - \left(\frac{1}{2}\right)^2 + 1\right) = \frac{9}{4} = 2.25 \\ y_3 &= \cdots = 3.375 \\ y_4 &= \cdots = 4.4375 \end{aligned}$$

Exact solution: $y(t) = (t+1)^2 - \frac{1}{2}e^t$. Then

$$y(1) = 4 - \frac{e}{2} \approx 2.6409 \implies |y(1) - y_2| \approx 0.39$$

Algorithm 6.1 Euler's Method

```
1: Input: End points $a$, $b$, number of intervals $N$, initial condition $y_0$.
2: procedure EULER($a$, $b$, $N$, $y_0$)
3:     Set $h = \frac{b-a}{N}$, $t_i = a$, $y_i = 0$
4:     for $i = 1$ to $N$ do
5:         Set $y_i = y_{i-1} + h\, f(t_i, y_i)$
6:         Set $t_i = t_{i-1} + h$
7:     end for
8: end procedure
9: Output: Approximation $y_i$ to $y(t_i)$ at the $N+1$ values $t_i$.
```

Definition 6.1 (Local truncation error)**.** Suppose we have an iterative method

$$y_{i+1} = y_i + h\,\phi(t_i, y_i), \qquad i = 0, 12, \dots$$

with initial condition y_0. The local truncation error is defined by

$$\tau_i = \frac{y(t_{i+1}) - y(t_i)}{h} - \phi(t_i, y(t_i))$$

In other words, we replace y_i with $y(t_i)$ and y_{i+1} with $y(t_{i+1})$ in the method and rearrange it. Note that for Euler's method $\phi(t,y) = f(t,y)$.

If $y_i = y(t_i)$, then

$$y(t_{i+1}) - y(t_i) = h\,\tau_{i+1}(h)$$

We have

$$y_{i+1} = y_i + h\,\phi(t_i, y_i)$$

so

$$\begin{aligned}
y(t_{i+1}) - y_{i+1} &= y(t_{i+1}) - y_i - h\,\phi(t_i, y_i) \\
&= h\left(\frac{y(t_{i+1}) - y_i}{h} - \phi(t_i, y_i)\right) \\
&= h\left(\frac{y(t_{i+1}) - y(t_i)}{h} - \phi(t_i, y(t_i))\right) = h\,\tau_{i+1}(h)
\end{aligned}$$

Remark 6.1. In general $y_i \neq y(t_i)$ because errors are accumulated from previous time steps.

Note

$\tau_{i+1} = \mathcal{O}(h)$ for Euler's method

6.2 High-Order Taylor Methods

Idea: Similar to Euler, but leave more terms in the Taylor expansion.

nth order Taylor method: Perform Taylor series expansion and truncate after $n+1$ terms:

$$\begin{aligned}
y(t_i + h) &= y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \cdots + \frac{h^n}{n!}y^{(n)}(t_i) + HOT \\
&= y(t_i) + h\sum_{j=1}^{n} \frac{h^{j-1}}{j!} y^{(j)}(t_i) + HOT
\end{aligned}$$

where HOT : denote high order terms.

Disregard HOT to obtain:

$$y_{i+1} = y_i + h\underbrace{\sum_{j=1}^{n} \frac{h^{j-1}}{j!} y^{(j)}(t_i)}_{:=\phi(t_i)} \qquad i = 0, 1, \dots, N-1$$

where $h = \frac{b-a}{n}$ and $y_0 = y(a)$.

As $y'(t) = f(t,y)$ we can find $y''(t)$ and $y'''(t)$.

If $n = 1$: we obtain Euler's method.

If $n = 2$:

$$y''(t) = \frac{d}{dt}f(t, y(t)) = \frac{\partial f}{\partial t} + \frac{\partial y}{\partial t}\frac{\partial f}{\partial y}$$

Then

$$y_{i+1} = y_i = h\left[f(t_i, y_i) + \frac{h}{2}\left(\frac{\partial f}{\partial t}(t_i, y_i) + \overbrace{f(t_i, y_i)}^{\frac{\partial y}{\partial t}(t_i, y_i)}\frac{\partial f}{\partial y}(t_i, y_i)\right)\right], \qquad i = 0, 1, \dots, N-1$$

Exercise 6.2. Consider again the IVP

$$\begin{aligned}
y' &= \underbrace{y - t^2 + 1}_{f(t,y)}, \qquad 0 \leq t \leq 2 \\
y(0) &= \frac{1}{2}
\end{aligned}$$

1. Show that $f'(t,y) = y - t^2 - 2t + 1$
2. Hence write down Taylor's method of order two.
3. Use Taylor's method of order 2 with $h = \frac{1}{2}$ to obtain:

$$y_1 \approx y\left(\frac{1}{2}\right), \qquad y_2 \approx y(1), \qquad \dots$$

4. Compare with Euler's method and exact solution ($y(t) = (t+1)^2 - \frac{1}{2}e^t$).

Solution

1.

$$y''(t) = \frac{d}{dt}f(t,y) = y' - 2t = y - t^2 + 1 - 2t$$

Also note:

$$\frac{\partial f}{\partial t} = -2t, \qquad \frac{\partial f}{\partial y} = 1$$

so

$$y'' = \frac{\partial f}{\partial t} + y'\frac{\partial f}{\partial y} = -2t + y' = y - t^2 + 1 - 2t$$

2.

$$y_{i+1} = y_i + h\left[y_i - t_i^2 + 1 + \frac{h}{2}\left(y_i - t_i^2 + 1 - 2t_i\right)\right], \qquad i = 0, 1, \dots, N-1$$

with $y_0 = \frac{1}{2}$.

3.

$$\begin{aligned} y_0 &= \frac{1}{2} \\ y_1 &= \cdots = 1.4375 \\ y_2 &= \cdots = 2.6797 \\ y_3 &= \cdots = 4.1045 \\ y_4 &= \cdots = 5.5135 \end{aligned}$$

Algorithm 6.2 second-order Taylor's Method

```
1: Input: End points $a$, $b$, number of intervals $N$, initial condition $y_0$, functions $f$, $\frac{\partial f}{\partial t}$, $\frac{\partial f}{\partial y}$
2: procedure EULER($a$, $b$, $N$, $y_0$)
3:     Set $h = \frac{b-a}{N}$, $t_i = a$, $y_i = 0$
4:     for $i = 1$ to $N$ do
5:         Set $y_i = y_{i-1} + h\,f(t_i, y_i)$
6:         Set $t_i = t_{i-1} + h\frac{h}{2}\left(\frac{\partial f}{\partial t}(t_i, y_i) + f(t_i, y_i)\frac{\partial f}{\partial y}(t_i, y_i)\right)$
7:     end for
8: end procedure
9: Output: Approximation $y_i$ to $y(t_i)$ at the $N+1$ values $t_i$.
```

Theorem 6.1. *If the nth order Taylor's method is used to approximate the solution to*

$$\begin{aligned} \frac{dy}{dt}(t) &= f(t, y(t)), \qquad a \le t \le b \\ y(a) &= y_0 \end{aligned}$$

with step size h and if $y \in C^{n+1}[a,b]$, then the local truncation error is $\mathcal{O}(h^n)$.

Proof. Recall (assuming $y(t_i) = y_i$):

$$\begin{aligned} \tau_i &= \frac{y(t_{i+1}) - \overbrace{y(t_i)}^{y_i}}{h} - \phi(t_i, \overbrace{y(t_i)}^{y_i}) \\ y_{i+1} &= y_i + h\phi(t_i, y_i) \end{aligned}$$

If $y \in C^{n+1}[a,b]$ then

$$y(t_{i+1}) = y(t_i) + h \underbrace{\sum_{j=1}^{n} \frac{h^{j-1}}{j!} y^{(j)}(t_i)}_{\phi(t_i, y(t_i))} + \frac{h^{n+1}}{(n+1)!} y^{(n+1)}(\xi_i), \qquad \xi \in [t_i, t_{i+1}]$$

Rearranging gives

$$\underbrace{\frac{y(t_{i+1}) - y(t_i)}{h} - \phi(t_i, y(t_i))}_{\tau_{i+1}} = \frac{h^n}{(n+1)!} y^{(n+1)}(\xi_i) = \mathcal{O}(h^n)$$

Note that:

$$|y^{(n+1)}(\xi_i)| \le \max_{t \in [t_i, t_{i+1}]} |y^{(n+1)}(t)| := M_i \qquad \forall\, \xi_i \in [t_i, t_{i+1}]$$

Hence:

$$\left| \frac{h^n}{(n+1)!} y^{(n+1)}(\xi_i) \right| \le \frac{M_i}{(n+1)!} h^n = \mathcal{O}(h^n)$$

□

6.3 Runge-Kutta methods

If we integrate the ODE $y' = f(t,y)$, between t_i and t_{i+1}, we have

$$\int_{t_i}^{t_{i+1}} y'(t)dt = \int_{t_i}^{t_{i+1}} f(t,y)dt.$$

Let's apply the trapezium quadrature rule:

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t,y)dt \approx \frac{h}{2}[f(t_i, y(t_i)) + f(t_{i+1}, y(t_{i+1}))]$$

The trapezoidal method uses the average of $f(t,y)$ at t_i and t_{i+1}. That is

$$y_{i+1} = y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, y_{i+1})]$$

Exercise: Show that the local truncation error is $\mathcal{O}(h^2)$ using Taylor expansion of $f(t_{i+1}, y(t_{i+1})) = y'(t_{i+1}) = y'(t_i + h)$

- We notice that y_{i+1} appears on both sides of the expression. The formula is implicit in y_{i+1}.
- The trapezoidal method is an example of an implicit method.
- A root-finding algorithm such as Newton's method is requires to solve the implicit equation.
- However, it has $\mathcal{O}(h^2)$ local truncation error, but does not require computation of higher-order derivatives, unlike high-order Taylor series methods.
- We would like to take the idea of the trapezoidal method and produce a scheme that
  - keeps the second-order convergence property
  - is explicit
- Consider the modified scheme:

$$y_{i+1} = y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, y^*_{i+1})] \qquad i = 0, 1 \ldots$$

- We should choose y^*_{i+1} so that it is easy to compute directly from y_i and is an approximation to $y(t_{i+1})$.
- A natural choice is to use the Euler method to find y^*_{i+1}:

$$y^*_{i+1} = y_i + h\, f(t_i, y_i)$$

- This method is called the modified Euler method or Heun's method.
- The modified Euler method can thus be written as the two-stage method:

$$\begin{aligned} y_{i+1}^* &= y_i + h\, f(t_i, y_i) \\ y_{i+1} &= y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, y_{i+1}^*)] \end{aligned}$$

6.3.1 The Midpoint Method

- Alternatively, let's apply the midpoint rule to the ODE:

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t,y)dt \approx h\, f(t_{i+1/2}, y(t_{i+1/2}))$$

  where $t_{i+1/2} = t_i + \frac{h}{2}$.

- This time, we replace $y(t_{i+1/2})$ with an approximation $\hat{y}_{i+1/2}$ so that

$$y_{i+1} - y_i = h\, f(t_{i+1/2}, \hat{y}_{i+1/2})$$

- Again, use Euler's method so that

$$\hat{y}_{i+1/2} = y_i + \frac{h}{2}\, f(t_i, y_i)$$

  We obtain the midpoint method

$$y_{i+1} = y_i + h\, f\left(t_{i+1/2}, \overbrace{y_i + \frac{h}{2}\, f(t_i, y_i)}^{\hat{y}_{i+1/2}}\right)$$

Exercise 6.3. Consider the IVP

$$\begin{aligned} y' &= y - t^2 + 1, \qquad 0 \le t \le 1 \\ y(0) &= \frac{1}{2} \end{aligned}$$

1. Choose: Midpoint Method or Modified Euler Method. Use $h = \frac{1}{2}$ to obtain y_1 and y_2.
2. Compare with Euler's Method: $y_1 = \frac{5}{4}$ and $y_2 = \frac{9}{4}$ and the exact solution $y(y) = (t+1)^2 - \frac{1}{2}e^t$, $y(\frac{1}{2}) \approx 1.426$ and $y(1) = 2.641$

Note: error with Euler at $t = 1$ is approximately 0.391

Solution

Midpoint method. $h = \frac{1}{2}$, $y_0 = \frac{1}{2}$. Note that $f(t_0, y_0) = f(0, \frac{1}{2}) = \frac{3}{2}$ so

$$\begin{aligned} y_1 &= y_0 + \frac{1}{2} f\Big(\frac{1}{4}, \frac{1}{2} + \frac{1}{4}\frac{3}{2}\Big) = \cdots = 1.40625 \\ y_2 &= y_1 + \frac{1}{2} f\Big(\frac{3}{4}, y_1 + \frac{1}{4}\, f\Big(\frac{1}{2}, y_1\Big)\Big) = \cdots = 2.5976... \end{aligned}$$

so

$$|y_2 - y_1| = |2.5976... - 2.641....| \approx 0.0432$$

Modified Euler method: $h = \frac{1}{2}$, $y_0 = \frac{1}{2}$

$$y_1 = \cdots = 1.375$$

$$y_2 = \cdots = 2.5156$$

so $|y_2 - y_1| \approx 0.1252$

Exercise 6.4. Show that the local truncation error of the Midpoint Method is of second order

Solution

Midpoint method:

$$y_{i+1} = y_i + h\,\underbrace{f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2} f(t_i, y_i)\right)}_{\phi(t_i,y_i)}$$

Then

$$\tau_i = \frac{y(t_{i+1}) - y(t_i)}{h} - \phi(t_i, y(t_i))$$

From Taylor series we have:

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \mathcal{O}(h^3) \tag{6.1}$$

Then,

$$\frac{y(t_{i+1}) - y(t_i)}{h} = y'(t_i) + \frac{h}{2}y''(t_i) + \mathcal{O}(h^2)$$

On the other hand, from Taylor's expansion of $\phi(t_i, y(t_i))$:

$$\begin{aligned}\phi(t_i, y(t_i)) &= f\left(t_i + \frac{h}{2}, y(t_i) + \frac{h}{2}f(t_i, y(t_i))\right)\\ &= \underbrace{f(t_i, y(t_i))}_{y'(t_i)} + \underbrace{\frac{h}{2}\frac{\partial f}{\partial t}(t_i, y(t_i)) + \frac{h}{2}f(t_i, y(t_i))\frac{\partial f}{\partial y}(t_i, y(t_i))}_{\frac{h}{2}y''(t_i)} + \mathcal{O}(h^2)\end{aligned}$$

Hence

$$\phi(t_i, y(t_i)) = y'(t_i) + \frac{h}{2}y''(t_i) + \mathcal{O}(h^2).$$

Subtracting the previous equation from Equation 6.1 yields:

$$\phi(t_i, y(t_i)) = \frac{y(t_{i+1}) - y(t_i)}{h} + \mathcal{O}(h^2)$$

and so

$$\tau_i = \frac{y(t_{i+1}) - y(t_i)}{h} - \phi(t_i, y(t_i)) = \mathcal{O}(h^2)$$

6.3.2 Ruge-Kutta Methods

The two previous methods can be written in the general form (with $s = 2$):

$$y_{i+1} = y_i + h\sum_{j=1}^{s} b_j K_j$$

where

$$\begin{aligned}K_1 &= f(t_i, y_i),\\ K_2 &= f(t_i + c_2 h, y_i + h(a_{21}K_1)),\\ &\vdots\\ K_s &= f(t_i + c_s h, y_i + h(a_{s1}K_1 + \cdots + a_{s,s-1}K_{s-1}))\end{aligned}$$

These are explicit, s-stage, Runge-Kutta Methods.

A Butcher Tableau provides all the information needed for an s-state runge Kutta method concisely

Table 6.1: Butcher Tableau

0	
c_2	a_{21}

$$
\begin{array}{c|ccccc}
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_s & a_{s1} & \cdots & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}
$$

1-stage Runge-Kutta: $K_1 = f(t_i, y_1)$, $b_1 = 1$ so $y_{i+1} = y_i + hbK_1 = y_i + hf(t_i, y_i)$

Modified Euler:

$$y_{i+1} = y_i + \frac{\overset{b_1=b_2=1/2}{\overbrace{h}}}{2} \left[\underbrace{f(t_i, y_i)}_{K_1} + \underbrace{f(t_i + \overset{c_2=1}{\overbrace{1}}\ h, y_i + \overset{a_{21}=1}{\overbrace{1}}\ h\, \overbrace{f(t_i, y_i))}^{K_1}}_{K_2} \right]$$

Midpoint method:

$$y_{i+1} = y_i + h\, f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2} f(t_i, y_i) \right)$$

$b_2 = 1$, $b_1 = 0$, $c_2 = \frac{1}{2}$ $a_{21} = \frac{1}{2}$

Heun's Method of order three:

$$y_{i+1} = y_i + \frac{h}{4}\left(f(t_i, y_i) + 3f\left(t_i + \frac{2h}{3}, y_i + \frac{2h}{3} f\left(t_i + \frac{h}{3}, y_i + \frac{h}{3} f)t_i, y_i) \right) \right) \right)$$

for $i = 0, 1, \ldots, N-1$. This is a 3-stage RK method with tableau:

Table 6.2: Butcher Tableau for RK4

0			
$\frac{1}{3}$	$\frac{1}{3}$		
$\frac{2}{3}$	0	$\frac{2}{3}$	
	$\frac{1}{4}$	0	$\frac{3}{4}$

RK4 Method:

The most populat RK method is this one of order four:

For i=0,1,...N-1, do:

$$
\begin{aligned}
K_1 &= f(t_i, y_i) \\
K_2 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2} K_1 \right) \\
K_3 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2} K_2 \right) \\
K_4 &= f\left(t_i + h, y_i + hK_3 \right) \\
y_{i+1} &= y_i + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)
\end{aligned}
$$

with the usual initial condition y_0.

> **Note**
>
> The above method is sometimes called the Runge-Kutta method.

Table 6.3: Butcher Tableau for RK4

0		
$\frac{1}{2}$	$\frac{1}{2}$	
$\frac{1}{2}$	0	$\frac{1}{2}$

$$\begin{array}{c|cccc}
1 & 0 & 0 & 1 & \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}$$

Let us compare different methods. What is a fair computational comparison?

- RK4 with $h = 2$, and Euler's method with $h = ?$
- Modified Euler's method with $h = 2$, and Midpoint method with $h = ?$

Table 6.4: Comparison

t_i	Exact	Euler $h = 0.025$	Modified Euler $h = 0.05$	RK4
0.0	0.5000000	0.5000000	0.5000000	0.5000000
0.1	0.6574145	0.6554982	0.6573085	0.6574144
0.2	0.8292986	0.8253385	0.8290778	0.8292983
0.3	1.0150706	1.0089334	1.0147254	1.0150701
0.4	1.2140877	1.2056546	1.2136079	1.2140869
0.5	1.4256394	1.4147264	1.4250141	1.4256384

RK4 is optimal in some sense:

Evaluations per step	2	3	4	$5 \leq n \leq 7$	$8 \leq n \leq 9$	$10 \leq n$
Best possible local truncation error	$\mathcal{O}(h^2)$	$\mathcal{O}(h^3)$	$\mathcal{O}(h^4)$	$\mathcal{O}(h^{n-1})$	$\mathcal{O}(h^{n-2})$	$\mathcal{O}(h^{n-3})$

6.3.3 High-Order Equations and Systems of Differential Equations

Problem: Solve he IVP for a system of ODEs:

$$\begin{aligned}
\frac{d\mathbf{y}}{dt} &= \mathbf{f}(t, \mathbf{y}), \qquad a \leq t \leq b \\
\mathbf{y}(a) &= \mathbf{y}_0
\end{aligned}$$

Here, $\mathbf{y}$ and $\mathbf{f}$ are m-component vector functions and $\mathbf{y}_0 \in \mathbb{R}^m$. Let $\mathbf{y} = \begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_m(t) \end{bmatrix}$.

In component form, the system becomes:

$$\begin{aligned}
\frac{dy_j(t)}{dt} &= f_j(t, \mathbf{y}), \qquad j = 1, 2, \dots, m \\
y_j(a) &= y_{0,j}.
\end{aligned}$$

To solve this you may apply any of the methods we have seen to each of the components of the system simultaneously. For example, Euler's method is:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\,\mathbf{f}(t_i, \mathbf{y}_i)$$

Exercise 6.5. Consider the second-order IVP:

$$\begin{aligned}
y'' + y' &= t + y \qquad 0 \leq t \leq 1 \\
y(0) &= 1, \\
y'(0) &= 0
\end{aligned}$$

1. Write this IVP as a first-order system.
2. Perform two steps of Euler's method with $h = \frac{1}{2}$.

Solution

Let $z = y'$, then we obtain the system:

$$\begin{aligned} y' &= z \\ z' &= t - y - z \\ y(0) &= 1, \\ z(0) &= 0 \end{aligned}$$

if $\mathbf{y} = \begin{bmatrix} y \\ z \end{bmatrix}$, then

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y})$$

where $\mathbf{f} = \begin{bmatrix} f_1(t, \mathbf{y}) \\ f_2(t, \mathbf{y}) \end{bmatrix} = \begin{bmatrix} z \\ t + y - z \end{bmatrix}$.
Euler's method:

$$\begin{aligned} y_{i+1} &= y_i + h\, f_1(t_i, y_i, z_i) = y_i + h\, z_i \\ z_{i+1} &= z_i + h\, f_2(t_i, y_i, z_i) = z_i + h\,(t_i + y_i - z_i) \end{aligned}$$

For part 2. let $h = 1/2$ so

$$\begin{aligned} y_1 &= 1 + \left(\frac{1}{2}\right)(0) = 1 \\ z_1 &= 0 + \frac{1}{2}(0 + 1 - 0) = \frac{1}{2} \end{aligned}$$

and

$$\begin{aligned} y_2 &= 1 + \left(\frac{1}{2}\right)\left(\frac{1}{2}\right) = 1.25 \\ z_2 &= \frac{1}{2} + \frac{1}{2}\left(\frac{1}{2} + 1 - \frac{1}{2}\right) = 1 \end{aligned}$$

6.3.4 Convergence of Euler method

Definition 6.2 (Lipschitz Condition)**.** Let $(t, y) \in D \subset \mathbb{R}^2$. A function $f : D \to \mathbb{R}$ satisfies the Lipschitz condition (in variable y) on D if there exists $L > 0$ such that:

$$|f(t, y_1) - f(t, y_2)| \le L|y_1 - y_2|$$

for all $(t, y_1) \in D$ and for all $(t, y_2) \in D$.

L is called the Lipschitz constant.

Note

We would also say that f is Lipschitz continuous in y if it satisfies the Lipschitz conditions.

Exercise 6.6. Let $f(t, y) = t|y|$, and

$$D = \Big\{(t, y)\ |1 \le t \le 2, \quad -3 \le y \le 4\Big\}$$

Show that f satisfies the Lipschitz condition.

Solution

$$\Big|f(t, y_1) - f(t, y_2)\Big| = \Big|t|y_1| - t|y_2|\Big| = |t|\ \overbrace{\Big||y_1| - |y_2|\Big|}^{\le|y_1-y_2|} \le \overbrace{|t|}^{\le 2}\ |y_1 - y_2| \le 2|y_1 - y_2|$$

hence f satisfies the Lipschitz condition with $L = 2$.

There is an easier way to show that a function satisfies a Lipschitz condition.

Theorem 6.2. *Suppose $f(t,y)$ is defined on a convex set $D \subset \mathbb{R}^2$.*

If a constant $L > 0$ exists with $\left|\frac{\partial f}{\partial y}(t,y)\right| \leq L$, for all $(t,y) \in D$, then f satisfies a Lipschitz condition on D in the variable y with Lipschitz constant L.

Convex set: A set for which a straight line connecting any two points in D is contained in D.

Exercise 6.7. Let $f(t,y) = 1 + t\sin(ty)$, $D = \{(t,y) \mid 0 \leq t \leq 2, \quad -\infty < y < \infty\}$

Show that f satisfies the Lipschitz condition. What is the smallest possible Lipschitz constant L

Solution

Method 1: Direct application of the definition

$$\begin{aligned} |f(t,y_1) - f(t,y_2)| &= |1 + t\sin(ty_1) - (1 + t\sin(ty_2))| \\ &= |t(\sin(ty_1) - \sin(ty_2))| = \overbrace{|t|}^{\leq 2}\,|\sin(ty_1) - \sin(ty_2)| \\ &\leq 2|\sin(ty_1) - \sin(ty_2)| \end{aligned} \tag{6.2}$$

From mean value theorem there exists $\xi \in (y_1, y_2)$ such that

$$f'(\xi) = \frac{f(y_2) - f(y_1)}{y_2 - y_1} \implies \overbrace{t\cos(t\xi)}^{\frac{d}{d\xi}\sin(t\xi)} = \frac{\sin(ty_1) - \sin(ty_2)}{y_1 - y_2}$$

and so

$$t\cos(t\xi)(y_1 - y_2) = \sin(ty_1) - \sin(ty_2)$$

which combined with Equation 6.2 gives

$$\begin{aligned} |f(t,y_1) - f(t,y_2)| &\leq 2|\sin(ty_1) - \sin(ty_2)| \\ &\leq 2|t\cos(t\xi)(ty_1 - ty_2)| \\ &\leq 2\,\overbrace{|t|}^{\leq 2}\,\underbrace{|\cos(t\xi)|}_{\leq 1}\,|y_1 - y_2| \\ &\leq 4|y_1 - y_2| \end{aligned}$$

so $L = 4$.
Method 2: Use Theorem 6.2. Note D is convex. Then

$$\left|\frac{\partial f}{\partial y}(t,y)\right| = |t^2\cos(ty)| = \overbrace{t^2}^{\leq 4}\,\overbrace{|\cos(ty)|}^{\leq 1} \leq 4$$

Hence $L = 4$ and the Lipschitz condition is satisfied.

Theorem 6.3 (Euler's method error bound)**.** *Suppose f is continuous and satisfies a Lipschitz condition (in y) with constant L on*

$$D = \{(t,y) \mid a \leq t \leq b \quad and \quad -\infty < y < \infty\}$$

and that a constant M exists with $|y''(t)| \leq M$ for all $t \in [a,b]$.

Let $y_0, y_1, \dots, y_N$ be the approximations generated by Euler's method, then for each $i = 0,1,2,\dots,N$,

$$|y(t_i) - y_i| \leq \frac{hM}{2L}\left(e^{L(t_i - a)} - 1\right) \tag{6.3}$$

Corollary 6.1. *Under the same conditions of Theorem 6.3, the forward Euler method converges as $h \to 0^+$*

Proof. Since $t_i \leq b$ for all $i = 0,1,\dots,N$, then

$$|y(t_i) - y_i| \leq \frac{hM}{2L}\left(e^{L(b-a)} - 1\right) = \mathcal{O}(h) \to 0^+ \qquad \text{as} \quad h \to 0^+$$

□

6.3.5 Results needed for the proof of Theorem 6.3

Lemma 6.1. *For all $x \geq -1$ and any positive m, we have $0 \leq (1+x)^m \leq e^{mx}$*

Proof. If $x \geq -1$ then $1 + x \geq 0$ and so $(1+x)^m \geq 0$. In addition,

$$e^x = 1 + x + \underbrace{\frac{x^2}{2} + \frac{x^3}{3!} + \ldots}_{\geq 0 \quad \text{if} \quad x \geq -1} \geq 1 + x$$

So $e^{mx} \geq (1+x)^m$ □

Lemma 6.2. *If $s > 0$, $t \geq 0$ and $\{a_i\}_{i=0}^{k}$ is a sequence satisfying*

- $a_0 \geq -\frac{t}{s}$
- $a_{i+1} \leq (1+s)a_i + t$ *for each* $i = 0, 1, 2, \ldots, k = 1$.

then

$$a_{i+1} \leq e^{(i+1)s}\left(a_0 + \frac{t}{s}\right) - \frac{t}{s}$$

Proof. In the special case $t = 0$

$$a_i \leq (1+s)a_i \leq (1+s)^2 a_{i-1} \leq \cdots \leq (1+s)^{i-1} a_0$$

and using Lemma **??** we have

$$a_i \leq (1+s)^{i-1} a_0 \leq e^{(i+1)s} a_0$$

For $t > 0$ we prove by induction:

For the base case: $i = -1$ we have

$$\underbrace{e^{(i+1)s}}_{=1}\left(a_0 + \frac{t}{s}\right) - \frac{t}{s} = a_0$$

and so $a_0 \leq a_0$.

Induction step:

Assume statement holds for some i:

$$a_i \leq e^{is}\left(a_0 + \frac{t}{s}\right) - \frac{t}{s}$$

Then

$$\begin{aligned} a_{i+1} &\leq (1+s)a_i + t \\ &\overset{\text{ind.step}}{\widetilde{\leq}} \ (1+s)\left(e^{is}\left(a_0 + \frac{t}{s}\right) - \frac{t}{s}\right) + t \\ &\overset{\text{prev lemma}}{\widetilde{\leq}} \ e^s e^i\left(a_0 + \frac{t}{s}\right) - \frac{t}{s} \\ &= e^{(i+1)s}\left(a_0 + \frac{t}{s}\right) - \frac{t}{s} \end{aligned}$$

Hence, by mathematical induction, the statement is true for all $i \geq -1$. □

of Theorem Theorem 6.3. Recall

$$y_{i+1} = y_i + h\, f(t_i, y_i) \tag{6.4}$$

and from Taylor series:

$$y(t_{i+1}) = y(t_i) + h\ \underbrace{f(t_i, y(t_i))}_{y'(t_i)} + \frac{h^2}{2} y''(\xi_i), \qquad \xi_i \in (t_i, t_{i+1}) \tag{6.5}$$

Subtracting Equation 6.4 from Equation 6.5 we have

$$\underbrace{y(t_{i+1})-y_{i+1}}_{e_{i+1}} = \underbrace{y(t_i)-y_i}_{e_i} + h\Big[f(t_i, y(t_i) - f(t_i, y_i)\Big] + \frac{h^2}{2}y''(\xi_i)$$

Let $e_i = y(t_i) - y_i$ (error at time t_i) for $0, 1 \dots, N$.

We now bound e_{i+1} by e_i:

$$\begin{aligned}
|e_{i+1}| &= \Big|e_i + h\Big[f(t_i, y(t_i) - f(t_i, y_i)\Big] + \frac{h^2}{2}y''(\xi_i)\Big| \\
&\leq |e_i| + h\underbrace{\Big|f(t_i, y(t_i) - f(t_i, y_i)\Big|}_{\leq L|y(t_i)-y_i| \quad \text{(Lips. cond)}} + \frac{h^2}{2}\underbrace{|y''(\xi_i)|}_{\leq M} \\
&\leq |e_i| + hL\underbrace{|y(t_i) - y_i|}_{|e_i|} + \frac{h^2 M}{2} \\
&= (1 + hL)|e_i| + \frac{h^2 M}{2}
\end{aligned}$$

let $s = hL$, $a_i = |e_i|$, and $t = \frac{h^2 M}{2}$. Then, the previous expression becomes

$$a_{i+1} \leq (1 + s)a_i + t$$

We can use Lemma 6.2 to conclude that

$$\begin{aligned}
|e_{i+1}| &\leq e^{(i+1)Lh}\Big(\underbrace{|e_0|}_{=0} + \underbrace{\frac{hM}{2L}}_{\frac{t}{s}}\Big) - \frac{hM}{2L} \\
&= \frac{hM}{2L}(e^{(i+1)hL} - 1) \qquad \forall i = -1, 0, \dots, N-1
\end{aligned}$$

Note that $t_{i+1} = a + (i+1)h$ thus $(i+1)h = t_{i+1} - a$. Hence, after a shift of indices we have the results from Equation 6.3. □

6.3.6 Effect of round-off error

Suppose at every step of the method, a small round-off error is introduced, so that

$$\begin{aligned}
\tilde{y}_0 &= y_0 + \delta_0 \\
\tilde{y}_{i+1} &= \tilde{y}_i + h\, f(t_i, \tilde{y}_i) + \delta_{i+1} \qquad \text{for each} \quad i = 0, 1, \dots, N-1.
\end{aligned}$$

Theorem 6.4. *If $|\delta_i|\delta$ for each $i = 0, 1, 2, \dots, N$, then*

$$|y(t_i) - \tilde{y}_i| \leq \frac{1}{L}\left(\frac{hM}{2} + \frac{\delta}{h}\right)\left(e^{L(t_i-a)} - 1\right) + |\delta_0|e^{L(t_i-a)}.$$

Note that

$$\lim_{h\to 0}\left(\frac{hM}{2} + \frac{\delta}{h}\right) \to \infty$$

In fact $h_{min} = \sqrt{\frac{2\delta}{M}}$, so we need to use values $h \geq h_{min}$.

Appendix A

Calculus and Functions

A.1 Results from Calculus

Theorem A.1 (Intermediate Value Theorem)**.** *If $f \in C[a,b]$ and K is between $f(a)$ and $f(b)$, then there exists $c \in (a,b)$ for which $f(c) = K$*

Theorem A.2 (Mean-value Theorem)**.** *If $f \in C[a,b]$ and f is differentiable on (a,b), then there exits $c \in (a,b)$ for which*

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

Definition A.1. Let $f \in C^k[a,b]$. The Taylor polynomial of f about a point x_0 is defined by

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

Theorem A.3 (Taylor's Theorem)**.** *Suppose $f \in C^k[a,b]$ that $f^{(k+1)}$ exists on $[a,b]$, and $x_0 \in [a,b]$. For every $x \in [a,b]$, there exists $\xi(x)$ between x_0 and x with*

$$f(x) = P_n(x) + \frac{f^{(k+1)}(x_0)}{(k+1)!}(x - x_0)^{k+1}.$$

$$g(x) = g(p) + g'(p)(x - p) + \frac{g''(\xi)}{2}(x - p)^2$$

for some ξ (possibly different from the one above) between p and x.

Theorem A.4 (Rolle's Theorem)**.** *Suppose $f \in C[a,b]$ and f is differentiable on (a,b). If $f(a) = f(b)$, then a number c in (a,b) exists with $f'(c) = 0$.*

Theorem A.5 (Generalised Rolle's Theorem)**.** *Suppose $f \in C^N[a,b]$ and suppose f has $N+1$ zeros in $[a,b]$, i.e.*

$$f(\hat{x}_0) = 0, \quad f(\hat{x}_1) = 0, \quad \ldots, \quad f(\hat{x}_N) = 0$$

for distinct $\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_N \in [a,b]$. Then, $f^{(N)}(\epsilon) = 0$ for some $\epsilon in (a.b)$.

Lemma A.1 (Fundamental Lemmas")**.**

$$(1) \quad \sum_{j=1}^{m} 1 = m, \qquad (2) \quad \sum_{j=1}^{m} j = \frac{m(m+1)}{2}, \qquad (3) \quad \sum_{j=1}^{m} j^2 = \frac{m(m+1)(2m+1)}{6}$$

A.2 Big O Notation

Definition A.2. Given two functions $f, g : \mathbb{N} \to \mathbb{R}$ we write $f(n) = \mathcal{O}(g(n))$ as $n \to \infty$, if there exists a constant $C > 0$ and $N \in \mathbb{N}$ such that

$$|f(n)| \leq Cg(n), \qquad \forall n > N.$$

We say f is Big O of g as $n \to \infty$.

Given two functions $f, g : \mathbb{R} \to \mathbb{R}$, we write $f(h) = \mathcal{O}(g(h))$ as $h \to 0$ if there exists $M > 0$ and $\delta > 0$ such that

$$|f(h)| \leq C|g(h)| \qquad \text{if} \quad |x| < \delta$$

We way f is Big-O of g as $h \to 0$.

A.3 Function Spaces

Definition A.3. Given an interval $[a, b] \subset \mathbb{R}$ we define

$$C[a, b] := \Big\{ g : [a, b] \to \mathbb{R} \;\Big|\; g \text{ is continuous} \Big\}$$

More generally we can define

Definition A.4.

$$C^k[a, b] := \Big\{ g : [a, b] \to \mathbb{R} \;\Big|\; g,\, g',\, g'',\, \dots,\, g^{(k)} \text{ is continuous} \Big\} \tag{A.1}$$

Remark A.1. We have

- C^{0}[a,b]:=C[a,b]
- $\cdots \subset C^2[a, b] \subset C^1[a, b] \subset C[a, b]$
- Terminology: $f \in C^k \implies$ "f is a C^k function" (or "f is of class C^k")
- We often say "f is sufficiently smooth". This means that $f \in C^k$ for large enough k for our results to hold.

Remark A.2. Given an open interval (a, b), we can define $C^k(a, b)$ analogous to $C^k[a, b]$.

Note that $C^k(a, b)$ and $C^k[a, b]$ are different.

- Functions (and their derivatives) in $C^k(a, b)$ can grow without bound as a and b are approached.
- On the contrary, functions (and their derivatives) in $C^k[a, b]$ are bounded over the whole of $[a, b]$.

Thus the notation C^k should only be used when there is no ambiguity.

Appendix B

Significant figures

Let us review the following definition:

Definition B.1. Let $p \in \mathbb{R}$ be represented in scientific notation as

$$p = \pm\alpha.\square\ldots\square\square\cdots \times 10^n$$

where $\alpha \in \{1,\ldots,9\}$. We say that an approximation $\tilde{p}$ to p has m **significant figures (or digits)** of p if

$$|p-\tilde{p}| \leq 5\times 10^{n-m}$$

i.e.

$$|p-\tilde{p}| \leq \overbrace{0.000\ldots0}^{m}\ 5\times 10^{n}$$

Proposition B.1. *If $\tilde{p}$ has m significant figures of p the (absolute) relative error satisfies*

$$\frac{|p-\tilde{p}|}{|p|} \leq 5\times 10^{-m} \tag{B.1}$$

Note that

$$\frac{|p-\tilde{p}|}{|p|} \leq \frac{5\times 10^{n-m}}{|p|} \leq \frac{5\times 10^{n-m}}{\alpha\times 10^{n}} = \frac{5}{\alpha}\times 10^{-m} \leq 5\times 10^{-m} \tag{B.2}$$

where we have used the fact that $1 \leq \alpha$ and so $\alpha^{-1} \leq 1$ as well as

$$|p| = |\alpha.\square\ldots\square\square\cdots\times 10^n| \geq \alpha\times 10^n$$

We now show that how $-\log_{10}$ of the relative error is an estimate of the number of significant figures. Suppose $\tilde{p}$ has m significant figures of p and that the relative error is

$$\frac{|p-\tilde{p}|}{|p|} \approx 10^{-M}.$$

Then,

$$|p-\tilde{p}| \approx 10^{-M}|p| \leq (\alpha+1)\times 10^{n-M} \leq 10\times 10^{n-M} = 10^{n-(M-1)} \leq 5\times 10^{n-(M-1)}$$

where we have used the fact that $|p| \leq (\alpha+1)\times 10^n$ and $\alpha \leq 9$. The previous statement means that $M-1 \leq m$, i.e. $\tilde{p}$ has at least $M-1$ significant figures of p. Note that if $m = M+1$ Equation B.2 would lead to a contradiction:

$$\frac{|p-\tilde{p}|}{|p|} \approx 10^{-M} \leq 5\times 10^{-m} = 5\times 10^{-M-1} = 0.5\times 10^{-M}$$

Hence $M-1 \leq m \leq M$. In conclusion,

$$m \leq \overbrace{-\log_{10}\left(\frac{|p-\tilde{p}|}{|p|}\right)}^{M} \leq m+1$$

and the $\log_{10}$ of the relative error provides an rough estimate of the significant digits of the approximation.