Class notes

Numerical Analysis

Fabio Nobile

(with modifications by Daniel Kressner)

Fall 2024

Last update: September 18, 2024

Table of contents

Chapter 1

Nonlinear Equations

In this chapter, we are interested in finding numerically the roots of a continuous function $f(x) : [a, b] \to \mathbb{R}$, also known as the values $\alpha \in [a, b]$ such as

$$f(\alpha) = 0. \tag{1.1}$$

For this purpose, we will study several numerical methods which allow us to find an approximate solution of (1.1)

1.1 Example: electrical circuit

Let us consider the electrical circuit shown in Figure 1.1 (left)

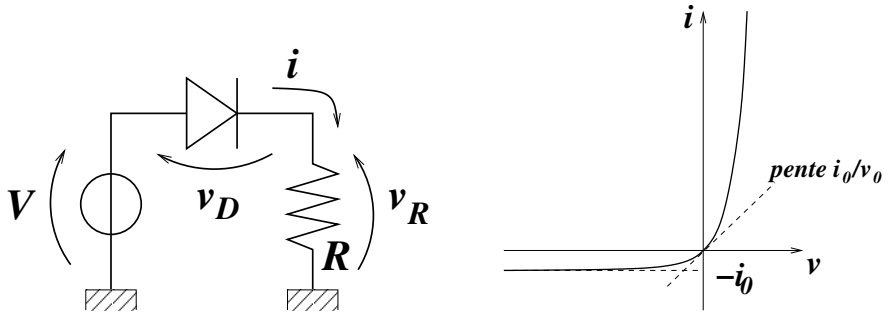


Figure 1.1: Left: Electrical circuit including a voltage generator, a resistance R and a standard diode.
Right: Current–voltage relationship of a standard diode.

which includes:

- A voltage generator that generates a constant voltage V.
- An electrical resistance R with a linear relationship $v_R = Ri$ between the current i and the voltage v_R.

- A standard diode. The current–voltage (I–V) relationship is given by the Shockley diode equation

$$i = i_0 \left(e^{v_D/v_0} - 1 \right),$$

where v_0 and i_0 are two constants that define the diode's behavior. This relationship is shown in Figure 1.1 (right).

To find the voltage across the diode, we combine the following expressions:

$$\begin{cases} v_R + v_D = V \\ v_R = Ri \\ i = i_0 \left(e^{v_D/v_0} - 1 \right) \end{cases} \qquad \Longrightarrow \qquad Ri_0 \left(e^{v_D/v_0} - 1 \right) + v_D = V.$$

The voltage v_D across the diode is therefore the root of the *nonlinear equation*

$$f(x) = 0, \qquad \text{where} \quad f(x) = Ri_0 \left(e^{x/v_0} - 1 \right) + x - V. \tag{1.2}$$

1.2 Bisection

Bisection (also called method of dichotomy) is based on the following observation:

Remark 1.1. *Consider a* continuous *function* $f : [a, b] \to \mathbb{R}$ *such that* $f(a)f(b) < 0$. *Then* f *has at least one root in* $[a, b]$.

Let us now consider the midpoint of the interval $[a, b]$: $x_m = \frac{a+b}{2}$. We then have the following three possibilities:

- If $f(x_m)f(a) < 0 \implies$ then a root exists in the interval $[a, x_m]$;
- If $f(x_m)f(b) < 0 \implies$ then a root exists in the interval $[x_m, b]$;
- If $f(x_m) \quad = 0 \implies$ we found a root of f.

In the first two cases, we can repeat this process for the sub-interval that is known to contain a root.

Algorithm 1.1: Bisection method (without stopping criterion)

We set $a^{(0)} = a$, $b^{(0)} = b$ and $x^{(0)} = \frac{a^{(0)}+b^{(0)}}{2}$;
for $k = 0, 1, \ldots$ **do**
  if $f(x^{(k)})f(a^{(k)}) < 0$ **then** `// new interval` $[a^{(k)}, x^{(k)}]$
    $a^{(k+1)} = a^{(k)}$, $b^{(k+1)} = x^{(k)}$;
  else `// new interval` $[x^{(k)}, b^{(k)}]$
    $a^{(k+1)} = x^{(k)}$, $b^{(k+1)} = b^{(k)}$;
  end
  $x^{(k+1)} = \frac{a^{(k+1)}+b^{(k+1)}}{2}$;
end

The algorithm shown above is still incomplete because we have to setup a stopping criterion for the iterations, which we will do below.

Error check

At the kth iteration, the root is inside the interval $I^{(k)} = [a^{(k)}, b^{(k)}]$ of length $|I^{(k)}| = \frac{(b-a)}{2^k}$. The root is approximated by the midpoint $x^{(k)} = \frac{a^{(k)}+b^{(k)}}{2}$. If α represents the actual root of f, the error at the kth step of the algorithm is bounded by

$$|\alpha - x^{(k)}| \leq \frac{1}{2}|I^{(k)}| = \left(\frac{1}{2}\right)^{k+1}(b-a). \tag{1.3}$$

If we want to find an approximation of the root with a prescribed tolerance tol, the bound (1.3) tells us that we can stop the iterations of Algorithm 1.1 when $|I^{(k)}| < 2 \cdot \mathsf{tol}$. Therefore, we shall perform $k_{\min}$ iterations of the algorithm so that

$$\left(\frac{1}{2}\right)^{k_{\min}+1}(b-a) \leq \mathsf{tol} \qquad \Longrightarrow \qquad k_{\min} > \log_2\left(\frac{b-a}{\mathsf{tol}}\right) - 1.$$

We note that the bound (1.3) is *guaranteed*, which means that if we perform $k_{\min}$ iterations, bisection is guaranteed to return an approximation with an error smaller than tol. Here is the complete algorithm that includes the stopping criterion:

Algorithm 1.2: Bisection method (with stopping criterion)

Data: $f(x)$, $[a, b]$, tol
Result: α (approximate root), niter (number of iterations)
$a^{(0)} = a$, $b^{(0)} = b$, $x^{(0)} = \frac{a^{(0)}+b^{(0)}}{2}$;
$k_{\min} = \lceil \log_2\left(\frac{b-a}{\mathsf{tol}}\right) - 1 \rceil$;
for $k = 0, 1, \ldots, k_{\min} - 1$ **do**
  if $f(x^{(k)})f(a^{(k)}) < 0$ **then** `// new interval` $[a^{(k)}, x^{(k)}]$
    $a^{(k+1)} = a^{(k)}$, $b^{(k+1)} = x^{(k)}$;
  else `// new interval` $[x^{(k)}, b^{(k)}]$
    $a^{(k+1)} = x^{(k)}$, $b^{(k+1)} = b^{(k)}$;
  end
  $x^{(k+1)} = \frac{a^{(k+1)}+b^{(k+1)}}{2}$;
end
$\alpha = x^{(k_{\min})}$, niter=$k_{\min}$.

Advantages and disadvantages

+ Once we find an interval where the (continuous) function changes sign, the algorithm is guaranteed to converge to a root.

+ We have a precise control on the error.

- If the function does not change sign around a root ($f(x) = x^2$), it is not possible to use this algorithm.

- The convergence of the algorithm is relatively slow; the error is divided only by two at every iteration.

Example 1.1 (Electrical circuit)**.** *Let us come back to the example of the electrical circuit from Section 1.1. Using* **Python***, we visualize the function* $f(x)$ *for* $i_0 = 1$, $v_0 = 0.1$, $R = 1$, $V = 1$.

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

i0 = 1
v0 = 0.1
R = 1
V = 1
f = lambda x: R * i0 * (np.exp(x / v0) - 1) + x - V
x = np.linspace(-0.2, 0.2, 40)
plt.plot(x, f(x), color="b", linewidth=2)
plt.plot(x, 0 * x, "--k", linewidth=2)
plt.grid(True)
```

Figure 1.2 shows the function $f(x)$*. There is obviously a root in the interval* $[-0.2, 0.2]$.

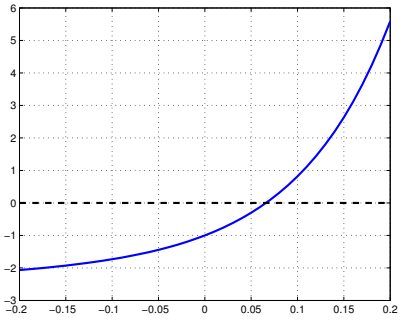


Figure 1.2: Plot of the function $f(x)$ from Eqn (1.2)

We apply the bisection method using the function bisection *from the Moodle page (call* help(bisection) *for details about the inputs and outputs of the function).*

```
import numpy as np
from functions import bisection

i0 = 1
```

```
v0 = 0.1
R = 1
V = 1
f = lambda x: R * i0 * (np.exp(x / v0) - 1) + x - V
zero, res, niter, inc, err = bisection(f, -0.2, 0.2, 1e-8, 10000)
print("zero " + str(zero))
print("res " + str(res))
print("iterations " + str(niter))

# OUTPUT
# zero 0.06596105694770812
# res 7.085173225895858e-08
# iterations 25
```

1.3 Fixed-point iterations

1.3.1 Introduction using the electrical circuit example

Let us consider again the electrical circuit from Section 1.1. To find an approximation of the root, we could follow the process below:

1st method

- Let us suppose that we have a good general idea about the voltage across the diode, which we name $v_D^{(0)}$. For example, if the diode is open, we would expect the voltage across the diode to be zero and we can set $v_D^{(0)} = 0$.

- Given the voltage $v_D^{(0)}$ across the diode, we can find the voltage across the resistance $v_R^{(0)} = V - v_D^{(0)}$ and thus the current $i^{(0)} = v_R^{(0)}/R = (V - v_D^{(0)})/R$.

- As the currents passing through the resistance and the diode are the same, we can find a new estimation of the voltage across the diode by reversing the characteristic curve of the diode:

$$v_D^{(1)} = v_0 \log\left(\frac{i^{(0)}}{i_0} + 1\right).$$

  We expect the new estimation $v_D^{(1)}$ to be better than the previous one.

We can repeat this process and find the second estimation $v_D^{(2)}$ and so on. We expect the sequence $v_D^{(0)}, v_D^{(1)}, v_D^{(2)}, \ldots$ to converge towards the "true" voltage of the diode. Formally, the method we just described takes the

following form: Given an approximation $v_D^{(k)}$ at the k^{th} iteration, we find the new estimation $v_D^{(k+1)}$ using

$$v_D^{(k+1)} = v_0 \log\left(\frac{V - v_D^{(k)}}{Ri_0} + 1\right). \tag{1.4}$$

Let us try (1.4) in `Python`:

```
import numpy as np

i0 = 1
v0 = 0.1
R = 1
V = 1
vD = 0
for i in range(10):
    vD = v0 * np.log((V - vD) / (R * i0) + 1)
    # formats to 15 decimal places
    formatted_string = "{:.15f}".format(vD)
    vD = float(formatted_string)
    print("vD = " + str(vD))

# OUTPUT
# vD = 0.069314718055995
# vD = 0.065787500825971
# vD = 0.065970026647302
# vD = 0.065960589502512
# vD = 0.065961077453676
# vD = 0.065961052224034
# vD = 0.065961053528539
# vD = 0.065961053461089
# vD = 0.065961053464577
# vD = 0.065961053464397
```

It can be seen that the sequence (1.4) converges to the same value we previously found using the bisection method. Moreover, after 10 iterations, 12 significant decimal figures appear to be "stable" and we expect the error to be smaller than 10^{-12}. This method seems to converge more quickly than bisection.

The sequence (1.4) is not the only way to set up a recursion. In fact, we could have defined another process:

2nd method

- Given an initial value of the voltage $v_D^{(0)}$ across the diode, we can find the current $i^{(0)}$ going through the diode: $i^{(0)} = i_0\left(e^{v_D^{(0)}/v_0} - 1\right)$.
- We then find the voltage across the resistance $v_R^{(0)} = Ri^{(0)}$.

- Finally, we can compute a new approximation of the voltage across the diode by $v_D^{(1)} = V - v_R^{(0)}$,

and so on. The complete procedure takes the following form: Given an estimation $v_D^{(k)}$ at the k^{th} iteration, we compute the new approximation $v_D^{(k+1)}$ by

$$v_D^{(k+1)} = V - Ri_0 \left(e^{v_D^{(k)}/v_0} - 1 \right). \tag{1.5}$$

Let us also try (1.5) in `Python`:

```
import numpy as np

i0 = 1
v0 = 0.1
R = 1
V = 1
vD = 0
for i in range(10):
    vD = V - R * i0 * (np.exp(vD / v0) - 1)
    # formats to 15 decimal places
    formatted_string = "{:.15f}".format(vD)
    vD = float(formatted_string)
    print("vD = " + str(vD))

# OUTPUT
# vD = 1.0
# vD = -22024.465794806718
# vD = 2.0
# vD = -485165193.4097903
# vD = 2.0
# vD = -485165193.4097903
# vD = 2.0
# vD = -485165193.4097903
# vD = 2.0
# vD = -485165193.4097903
```

Even though the motivation for the recursion (1.5) is as reasonable as the one for (1.4), it does not seem to converge at all. This means that we obviously cannot use it to find the voltage across the diode.

Let us now try to recapitulate and formalize everything that we did up to now. We had to solve the following equation:

$$f(x) = Ri_0 \left(e^{x/v_0} - 1 \right) + x - V = 0.$$

For the first method, we rewrote this equation under its equivalent form

$$\begin{aligned} &\text{Current through the diode:} && i_0\left(e^{x/v_0}-1\right)=\frac{V-x}{R}\\ &\text{Voltage across the diode:} && x=v_0\log\left(\frac{V-x}{Ri_0}+1\right). \end{aligned}$$

Then we performed the iteration $x^{(k+1)}=v_0\log\left(\frac{V-x^{(k)}}{Ri_0}+1\right)$.

For the second method, we rewrote the equation simply under the form:

$$x=V-Ri_0\left(e^{x/v_0}-1\right)$$

and we performed the iterations $x^{(k+1)}=V-Ri_0\left(e^{x^{(k)}/v_0}-1\right)$.

In both cases, we rewrote the nonlinear equation $f(x)=0$ under an equivalent form

$$x=\phi(x), \tag{1.6}$$

leading to the iterative method

$$x^{(k+1)}=\phi(x^{(k)}),\quad k=0,1,\ldots. \tag{1.7}$$

An equation of the form (1.6) is called *fixed-point equation* because the value α that satisfies the equation: $\alpha=\phi(\alpha)$ has the property that ϕ applied to α does not change the value of α. This value is therefore called a *fixed point* of the function ϕ.

The iterative process (1.7) is called *fixed-point method* or *fixed-point iteration*.

1.3.2 Fixed-point method

Given a nonlinear equation $f(x)=0$, the fixed-point method consists in rewriting, under an equivalent form, the equation $f(x)=0$ as a fixed-point equation $x=\phi(x)$, so that if α is the root of f

$$f(\alpha)=0 \qquad \Longrightarrow \qquad \alpha=\phi(\alpha).$$

Algorithm 1.3: Fixed-point method (without stopping criterion)

Choose a starting point $x^{(0)}$ sufficiently close to a fixed point α. **for** $k=0,1,\ldots$ **do**
  $x^{(k+1)}=\phi(x^{(k)})$
end

Graphical interpretation

The fixed-point method admits a graphical interpretation. The solution of the fixed-point equation $x = \phi(x)$ can be viewed as the solution of the system

$$\begin{cases} y = \phi(x), \\ y = x. \end{cases}$$

Graphically, this means that the fixed points of ϕ are given by the intersection between the function $y = \phi(x)$ and the line $y = x$.

In the same manner, the fixed-point method starts from a value $x^{(k)}$ and first computes $y^{(k+1)} = \phi(x^{(k)})$ and then $x^{(k+1)} = y^{(k+1)}$. Therefore, the value $x^{(k+1)}$ on the abscissa has the same distance from the origin as the value $y^{(k+1)}$ on the ordinate axis. Let us now consider the equation

$$f(x) = x + \log(x+1) - 2 = 0$$

and the 4 equivalent fixed-point equations

$$\begin{aligned} x &= \phi_1(x) = x - \frac{1}{2}(\log(x+1) + x - 2), \\ x &= \phi_2(x) = 2 - \log(x+1), \\ x &= \phi_3(x) = e^{2-x} - 1, \\ x &= \phi_4(x) = \frac{1}{2}x(\log(x+1) + x), \end{aligned} \tag{1.8}$$

We can notice from Figure 1.3 that the fixed-point methods for ϕ_1 and ϕ_2 converge whereas the fixed-point methods for ϕ_3 and ϕ_4 do *not* converge.

Convergence analysis

A detailed study of the results in Figure 1.3 reveals that the fixed-point method converges when the slope of the function ϕ at the fixed point, $\phi'(\alpha)$, is in the interval $(-1, 1)$ whereas it diverges when $|\phi'(\alpha)| > 1$. This conclusion holds at least for all initial points $x^{(0)}$ in the interval $[0, 5]$. For initial points further from the fixed point, this conclusion can be false.

Let us establish more rigorously this result. We are interested in studying the behavior of the error $e^{(k)} = |x^{(k)} - \alpha|$ at the k^{th} iteration, at least when $x^{(k)}$ is not too far away from α. For this purpose, we recall the Taylor series of 1^{st} order of ϕ around the fixed point α:

$$\phi(x) = \underbrace{\phi(\alpha)}_{=\alpha} + \phi'(\alpha)(x - \alpha) + R(x).$$

The remainder term satisfies

$$R(x) = o(|x - \alpha|) \quad \Longleftrightarrow \quad \lim_{x \to \alpha} \frac{R(x)}{|x - \alpha|} = 0,$$

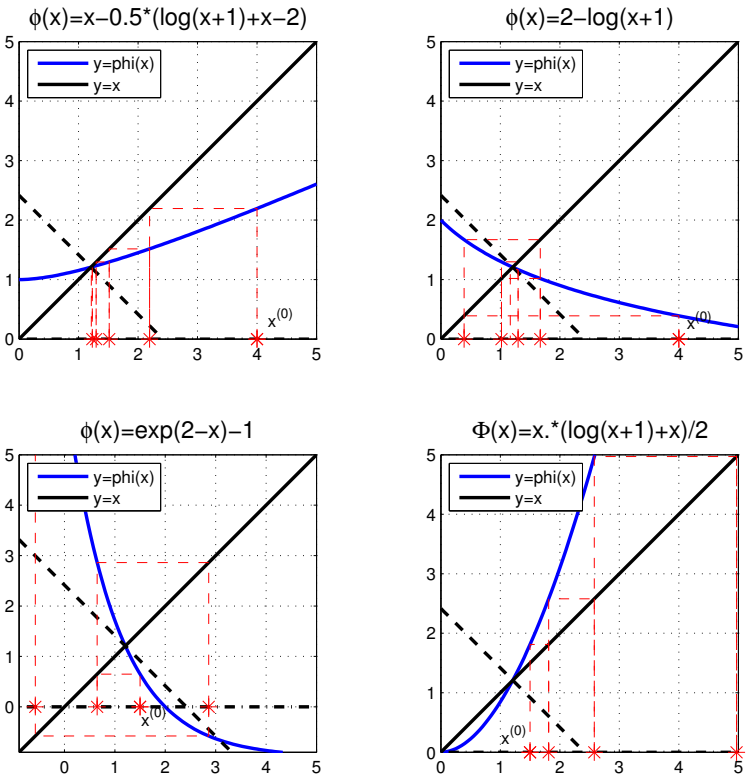


Figure 1.3: First 5 iterations of the fixed-point method using one of the four different fixed-point equations (1.8). The dashed line is the line with slope -1 through the fixed-point.

that is, the function $R(x)$ approaches zero more quickly than the function $|x-\alpha|$, as $x \to \alpha$. This allows us to approximate the error at the first iteration as follows:

$$x^{(1)} - \alpha = \phi(x^{(0)}) - \phi(\alpha) = \phi'(\alpha)(x^{(0)} - \alpha) + \underbrace{o(|x^{(0)} - \alpha|)}_{\text{small term}}$$

and, hence,

$$e^{(1)} = |\phi'(\alpha)|e^{(0)} + \underbrace{o(e^{(0)})}_{\text{small term}} .$$

The term $o(e^{(0)})$ is negligible compared to the term $|\phi'(\alpha)|e^{(0)}$ when $x^{(0)}$ is sufficiently close to α. We can now iterate the previous reasoning

$$\begin{aligned} e^{(k)} &= |\phi'(\alpha)|e^{(k-1)} + \text{small term} \\ &= |\phi'(\alpha)|^2 e^{(k-2)} + \text{small term} \\ &\vdots \\ &= |\phi'(\alpha)|^k e^{(0)} + \text{small term.} \end{aligned}$$

We conclude that *the error $e^{(k)}$ approaches zero if $|\phi'(\alpha)| < 1$* as we already guessed graphically. The result we just showed is summarized in the following theorem:

Theorem 1.1. *Consider a function $\phi : [a, b] \to \mathbb{R}$ of class C^1 (continuously differentiable) and a fixed point $\alpha \in (a, b)$ of ϕ.*
If

$$|\phi'(\alpha)| < 1$$

then *there exists $\delta > 0$ such that the fixed-point iteration $x^{(k+1)} = \phi(x^{(k)})$ converges to α for every $x^{(0)} \in [\alpha - \delta, \alpha + \delta]$. Moreover,*

$$\lim_{k\to\infty} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|} = |\phi'(\alpha)|.$$

Error control

Algorithm 1.3 is not complete as we need to add a stopping criterion (otherwise, we end up with infinite loops!).

Ideally, one would like to finish the iterations when the error $e^{(k)} = |x^{(k)} - \alpha|$ is smaller than a given tolerance. Sadly, we cannot use this criterion as we do not know the exact solution. Therefore, we will have to proceed differently.

Suppose that we computed the iteration $x^{(k)}$. If $x^{(k)}$ was a fixed point of ϕ, we would have $x^{(k)} = \phi(x^{(k)})$. It is likely that this is not true and $x^{(k)} - \phi(x^{(k)}) \neq 0$. This mismatch is measured by the *residual*

$$r^{(k)} = |x^{(k)} - \phi(x^{(k)})|.$$

We expect the residual to be small if $x^{(k)}$ is close to a fixed point α and it can be used as an *error indicator*. This motivates the

$$\textbf{stopping criterion}: \quad |x^{(k)} - \phi(x^{(k)})| \leq \mathsf{tol}.$$

Here is the complete version of the fixed-point method that includes a stopping criterion:

Algorithm 1.4: Fixed-point method with stopping criterion

Data: ϕ, $x^{(0)}$, tol
Result: α, res, niter
$r^{(0)} = \mathsf{tol} + 1$; $k = 0$;
while $r^{(k)} > \mathsf{tol}$ **do**
  $x^{(k+1)} = \phi(x^{(k)})$;
  $r^{(k)} = |x^{(k+1)} - x^{(k)}|$;
  $k = k + 1$;
end
$\alpha = x^{(k)}$, res=$r^{(k)}$, niter=k ;

Algorithm 1.4 stops at the k^{th} iteration for which $r^{(k)} \leq \mathsf{tol}$ is satisfied for the first time. Usually, $r^{(k)} \leq \mathsf{tol}$ does *not* guarantee that the *true error* $|x^{(k)} - \alpha|$ is also smaller than tol but *often* it is of the same order of magnitude. This is a consequence of the following result:

$$\begin{aligned} x^{(k)} - \alpha &= x^{(k)} - \phi(x^{(k)}) + \phi(x^{(k)}) - \alpha \\ &= x^{(k)} - \phi(x^{(k)}) + \phi(x^{(k)}) - \phi(\alpha) \\ &= x^{(k)} - \phi(x^{(k)}) + \phi'(\xi^{(k)})(x^{(k)} - \alpha), \end{aligned}$$

where $\xi^{(k)}$ is a suitable point of the interval $[\alpha, x^{(k)}]$ (implied by the mean value theorem). It follows that

$$|x^{(k)} - \alpha| = \frac{1}{|1 - \phi'(\xi^{(k)})|} r^{(k)}.$$

When the residual is smaller than tol, this implies for the error that

$$e^{(k)} \leq \frac{1}{|1 - \phi'(\xi^{(k)})|} \mathsf{tol}.$$

If $\phi'(\xi^{(k)})$ is not close to 1 then $e^{(k)} \lesssim \mathsf{tol}$, whereas if $\phi'(\xi^{(k)}) \approx 1$ then the true error can be a lot larger than the residual, thus making our error control unreliable. Note that $\phi'(\xi^{(k)}) \xrightarrow{k\to\infty} \phi'(\alpha)$ if the method converges. This once again shows the importance of the value of the slope ϕ at the fixed point.

1.3.3 Higher-order methods

Theorem 1.1 predicts fast convergence *close to the fixed point* when $|\phi'(\alpha)|$ is small. Note that $\phi'(\alpha) = 0$ does not imply that the method converges immediately; it just implies that the method converges very quickly as $x^{(k)}$ approaches α. The following definition of *order* provides a qualitative characterization of the speed of convergence close to α.

Definition 1.1. *Let α be a fixed point of ϕ. The fixed-point method $x^{(k+1)} = \phi(x^{(k)})$ is said to be* of order p *if the following holds: Whenever the sequence $\{x^{(k)}\}$ converges to α, there exists $C > 0$ such that*

$$\lim_{k\to\infty} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|^p} = C.$$

For the method to be of order 1, then it is required that $C < 1$.

The methods we studied for the functions ϕ_1 and ϕ_2 in Equation (1.8) are first-order methods. We can check this by using `Python`. As we do not know the exact solution, we use the result

$$|x^{(k)} - \alpha| = \frac{1}{|1 - \phi'(\xi^{(k)})|} r^{(k)}$$

and then

$$\lim_{k\to\infty} \frac{r^{(k+1)}}{(r^{(k)})^p} = \lim_{k\to\infty} \frac{|1 - \phi'(\xi^{(k+1)})|}{|1 - \phi'(\xi^{(k)})|^p} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|^p} = \frac{1}{|1 - \phi'(\alpha)|^{p-1}} C \neq 0.$$

So we check if the limit $\lim_{k\to\infty} \frac{r^{(k+1)}}{(r^{(k)})^p}$ approaches a constant (nonzero) for $p = 1$. For this purpose, we use the function fixed_point available on Moodle:

```
import numpy as np
from functions import fixed_point

phi = lambda x: x - 0.5 * (np.log(x + 1) + x - 2)
x0 = 4
tol = 1e-3
nmax = 1000
x, res, niter = fixed_point(phi, x0, tol, nmax)
print("x ")
print(x)
print("res")
print(res)
print("Number of iterations")
print(niter)
# Prints ratio of res:
print("Ratio of RES")
```

```
print(res[1:] / (res[:-1] ** 2))

# OUTPUT
# x
# [4.         2.19528104 1.516803   1.29690678 1.23267172 1.21473639
#  1.2098015  1.2084494 ]
# res
# [1.80471896e+00 6.78478046e-01 2.19896221e-01 6.42350574e-02
#  1.79353340e-02 4.93488414e-03 1.35209708e-03 3.70023111e-04]
# Number of iterations
# 7
# Ratio of RES
# [  0.20831313   0.47769002   1.3284236    4.34675574  15.34115371
#   55.52057452 202.40120959]
```

We can see that the value of $r^{(k+1)}/r^{(k)}$ approaches a constant for $p = 1$, whereas it diverges for $p = 2$.

According to Definition 1.1, for a method of order p, we have

$$|x^{(k+1)} - \alpha| \approx C|x^{(k)} - \alpha|^p$$

when $x^{(k)}$ is sufficiently close to α. Let us illustrate the benefit of having a method of order higher than 1 with a simple example: Given an initial error $|x^{(0)} - \alpha| = 10^{-1}$, consider two fixed-point methods: the first of order 1 with $C = 0.5$ and the second of order 2 with $C = 1$. Then the errors are *approximately* determined by the values in the table below:

	First order method ($C = 0.5$)	Second order method ($C = 1$)
$e^{(0)} = \lvert x^{(0)} - \alpha\rvert$	0.1	0.1
$e^{(1)} = \lvert x^{(1)} - \alpha\rvert$	0.05	10^{-2}
$e^{(2)} = \lvert x^{(2)} - \alpha\rvert$	0.025	10^{-4}
$e^{(3)} = \lvert x^{(3)} - \alpha\rvert$	0.0125	10^{-8}

From the table, it is clear that the error of the second order method decreases a lot faster than the error of the first order method!

We now aim at understanding under which conditions a method has an order greater than 1. This will help us develop new high order methods. For this purpose, let us come back to the approximation error of a fixed-point method:

$$x^{(k+1)} - \alpha = \phi(x^{(k)}) - \phi(\alpha) = \phi'(\alpha)(x^{(k)} - \alpha) + \underbrace{o(|x^{(k)} - \alpha|)}_{\text{small term}}.$$

What happens if $\phi'(\alpha) = 0$? To gain more insight into this case, we use a Taylor series of second order:

$$x^{(k+1)} - \alpha = \phi(x^{(k)}) - \phi(\alpha) = \underbrace{\phi'(\alpha)(x^{(k)} - \alpha)}_{=0} + \frac{1}{2}\phi''(\alpha)(x^{(k)} - \alpha)^2 + \underbrace{o(|x^{(k)} - \alpha|^2)}_{\text{small term}}.$$

If we neglect the "small" term, we can say that

$$x^{(k+1)} - \alpha \approx \frac{1}{2}\phi''(\alpha)(x^{(k)} - \alpha)^2$$

when $x^{(k)}$ is close to α. Hence, the method has order 2.

More generally, if $\phi'(\alpha) = \phi''(\alpha) = \ldots = \phi^{p-1}(\alpha) = 0$ and $\phi^p(\alpha) \neq 0$, the method has order p. The following theorem collects our considerations.

Theorem 1.2. *Consider a function $\phi : [a, b] \to \mathbb{R}$ of class C^p (p times continuously differentiable) and a fixed point $\alpha \in (a, b)$ of ϕ.* ***If***

$$\phi'(\alpha) = \phi''(\alpha) = \ldots = \phi^{p-1}(\alpha) = 0 \quad \textit{and} \quad \phi^p(\alpha) \neq 0, \;\; p \geq 2,$$

then *there exists $\delta > 0$ such that the iterations $x^{(k+1)} = \phi(x^{(k)})$ converge to α for every $x_0 \in [\alpha - \delta, \alpha + \delta]$. Moreover,*

$$\lim_{k\to\infty} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|^p} = \frac{1}{p!}|\phi^p(\alpha)|,$$

that is, the fixed-point method is of order p.

In the next section, we will study a second-order method, the Newton method.

Error control

Let us recall the previously established link between the error and the residual:

$$|x^{(k)} - \alpha| = \frac{1}{|1 - \phi'(\xi^{(k)})|} r^{(k)}.$$

For an order p method with $p > 1$ we have $\phi'(\alpha) = 0$ and in this case

$$|x^{(k)} - \alpha| \approx r^{(k)}.$$

Hence, for a method of order 2 or larger, the residual is an excellent approximation of the error when $x^{(k)}$ is close to α.

1.4 Newton Method

The Newton method is one of the most popular methods for solving a nonlinear equation $f(x) = 0$. Suppose that α is the desired root of f and we start from an initial value $x^{(0)}$ (sufficiently close to α). Then the first-order Taylor expansion of f around α gives

$$f(\alpha) = f(x^{(0)}) + f'(x^{(0)})(\alpha - x^{(0)}) + \text{"small term"}.$$

If we neglect the "small term" and take into account the fact that $f(\alpha) = 0$, we arrive at

$$f(x^{(0)}) + f'(x^{(0)})(\alpha - x^{(0)}) \approx 0.$$

A new (and hopefully better) approximation of α is provided by taking the root of this linear equation:

$$x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})}.$$

By repeating this process, we find $x^{(2)}$, and so on.

Algorithm 1.5: Newton method (without stopping criterion)

Given f, f' and $x^{(0)}$;
for $k = 0, 1, \ldots$ **do**
  $x^{(k+1)} = x^{(k)} - \dfrac{f(x^{(k)})}{f'(x^{(k)})}$
end

Figure 1.4 provides a graphical interpretation of the method. From the initial value $x^{(0)}$, we approach the curve $f(x)$ by the tangent line in $x^{(0)}$ and we find the root of the tangent line.

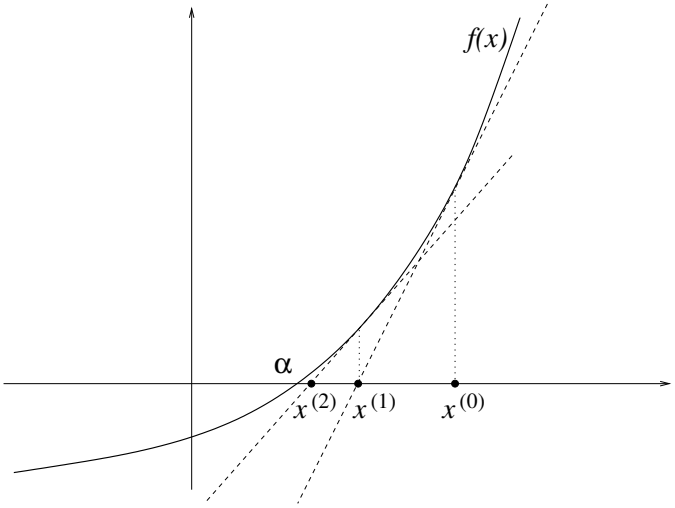


Figure 1.4: Graphical illustration of the Newton method.

Convergence analysis

We first notice that Algorithm 1.5 can be viewed as a fixed-point method:

$$x^{(k+1)} = \phi(x^{(k)}) \quad \text{where} \quad \phi(x) = x - \frac{f(x)}{f'(x)}.$$

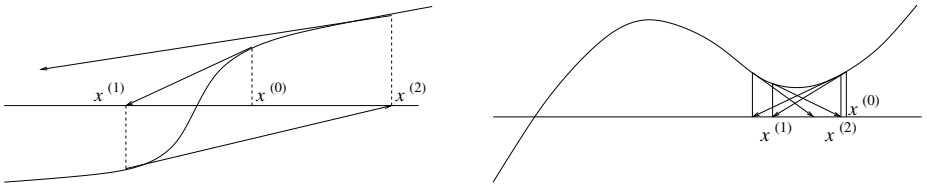


Figure 1.5: Two cases of non-convergence of the Newton method if the initial value is too far from the root.

To understand the order of this method, we need to determine the derivatives of ϕ at α. Computing the first and second derivatives gives

$$\phi'(x) = \frac{f(x)f''(x)}{(f'(x))^2},$$
$$\phi''(x) = \frac{f'(x)^2 f''(x) + f(x)f'(x)f'''(x) - 2f(x)f''(x)^2}{(f'(x))^3},$$

and thus

$$\phi'(\alpha) = 0 \qquad \text{if } f'(\alpha) \neq 0,$$
$$\phi''(\alpha) = \frac{f''(\alpha)}{f'(\alpha)} \neq 0 \qquad \text{if } f''(\alpha) \neq 0.$$

In the exceptional situation $f'(\alpha) = 0$, the Newton method is of first order at best. On the other hand, if $f'(\alpha) \neq 0$, the Newton method has (at least) second order.

Theorem 1.3. *Consider a function f of class C^2 and a root α of f. if $f'(\alpha) \neq 0$ and $f''(\alpha) \neq 0$, the Newton method converges with order two for every $x^{(0)}$ sufficiently close to α. Moreover*

$$\lim_{k\to\infty} \frac{|x^{(k+1)} - \alpha|}{|x^{(k)} - \alpha|^2} = \frac{1}{2}\frac{|f''(\alpha)|}{|f'(\alpha)|}.$$

It is important to notice that the convergence established by Theorem 1.3 is only *local*, that is, the convergence is guaranteed only if the initial value $x^{(0)}$ is sufficiently close to the root α. Figure 1.5 shows two cases of non-convergence of the Newton method if the initial value is too far from the root.

Error control

In Section 1.3.3 we have seen that controlling the error of a second-order method using the increment $|x^{(k+1)} - x^{(k)}|$ (residual of the corresponding

fixed-point equation) is very reliable if $x^{(k)}$ is sufficiently close to α. Thus, we can use the following stopping criterion for the Newton method

$$|x^{(k+1)} - x^{(k)}| \leq \mathsf{tol}.$$

The complete algorithm is given below:

Algorithm 1.6: Newton method (with stopping criterion)

Data: f, f', $x^{(0)}$, tol, nmax
Result: α, res, niter
$r^{(0)} = \mathsf{tol} + 1$; $k = 0$;
while $r^{(k)} > \mathsf{tol}$ *AND* $k < nmax$ **do**
  $x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$;
  $r^{(k+1)} = |x^{(k+1)} - x^{(k)}|$;
  $k = k + 1$;
end
$\alpha = x^{(k)}$, res=$r^{(k)}$, niter=k ;

Example 1.2 (Electrical circuit)**.** *Let us come back to the electrical circuit from Section 1.1 and let us solve the Equation* (1.2) *using the Newton method with the initial value* $x^{(0)} = 0.1$*. For this purpose, we use the function* newton *available on Moodle:*

```
import numpy as np
from functions import newton

i0 = 1
v0 = 0.1
R = 1
V = 1
f = lambda x: R * i0 * (np.exp(x / v0) - 1) + x - V
df = lambda x: R * i0 * np.exp(x / v0) / v0 + 1
x0 = 0
zero, res, niter, inc = newton(f, df, x0, 1e-8, 100000)
print("Results")
print("zero = " + str(zero))
print("residual = " + str(res))
print("number of iterations = " + str(niter))

# OUTPUT
# Results
# zero = 0.06596105346440571
# residual = 3.552713678800501e-15
# number of iterations = 5
```

It is clear here that the Newton method converges much faster than the fixed-point method (1.4)*. The function* newton *also returns us the list of*

increments $|x^{(k+1)} - x^{(k)}|$ *in the variable* inc. *This provides us the possibility of verifying the order of convergence*

```
import numpy as np
from functions import newton

i0 = 1
v0 = 0.1
R = 1
V = 1
f = lambda x: R * i0 * (np.exp(x / v0) - 1) + x - V
df = lambda x: R * i0 * np.exp(x / v0) / v0 + 1
x0 = 0
zero, res, niter, inc = newton(f, df, x0, 1e-8, 100000)
print("convergence")
print("---------------------------------")
for i in range(1, 5):
    print("order " + str(i))
    print(inc[1:] / inc[:-1] ** i)

# OUTPUT
# order 1
# [2.44095936e-01 1.22648213e-01 1.31765451e-02 1.70533784e-04]
# order 2
# [2.68505529 5.5270496  4.8414167  4.75532045]
# order 3
# [2.95356082e+01 2.49072339e+02 1.77886658e+03 1.32601717e+05]
# order 4
# [3.24891691e+02 1.12242579e+04 6.53603376e+05 3.69758790e+09]
```

We observe that the ratio $|x^{(k+1)} - x^{(k)}|/|x^{(k)} - x^{(k-1)}|^2$ *for* $k = 1, 2, 3, 4$ *is stable around* 5 *whereas the ratio* $|x^{(k+1)} - x^{(k)}|/|x^{(k)} - x^{(k-1)}|$ *approaches zero, which confirms that the method is of second order.*

1.5 Systems of nonlinear equations

In this section, we will generalize the Newton method to a system of nonlinear equations:

$$\begin{cases} f_1(x_1, \ldots, x_n) = 0, \\ f_2(x_1, \ldots, x_n) = 0, \\ \quad \vdots \\ f_n(x_1, \ldots, x_n) = 0. \end{cases}$$

For this purpose, let us introduce the compact notation

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{bmatrix}.$$

We call $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_n]^\top \in \mathbb{R}^n$ a root of $\mathbf{f}$ if it satisfies $\mathbf{f}(\boldsymbol{\alpha}) = \mathbf{0}$.

Example 1.3. *Let us consider the system of two equations in the two unknowns x_1, x_2:*

$$\begin{cases} f_1(x_1, x_2) = x_1^2 + x_1 x_2 - 10 = 0, \\ f_2(x_1, x_2) = x_2 + 3x_1 x_2^2 - 57 = 0. \end{cases} \tag{1.9}$$

The first equation implicitly defines the curve

$$f_1(x_1, x_2) = x_1^2 + x_1 x_2 - 10 = 0 \qquad \Longrightarrow \qquad x_2 = g_1(x_1) = \frac{10 - x_1^2}{x_1},$$

whereas the second equation defines the curve

$$f_2(x_1, x_2) = x_2 + 3x_1 x_2^2 - 57 = 0 \qquad \Longrightarrow \qquad x_1 = g_2(x_2) = \frac{57 - x_2}{3x_2^2}.$$

Figure 1.6 shows the two curves $x_2 = g_1(x_2)$ and $x_1 = g_2(x_2)$ plotted in `Python`*. Their intersection corresponds to the root of the system* (1.9).

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

g1 = lambda x1: (10 - x1**2) / x1
g2 = lambda x2: (57 - x2) / (3 * x2**2)
# plot de x2=g1(x1)
x1 = np.linspace(1, 3, 50)
plt.plot(x1, g1(x1), "b", linewidth=2)
# plot de x1=g2(x2)
x2 = np.linspace(1, 4.5, 50)
plt.plot(g2(x2), x2, "r", linewidth=2)
plt.legend([r"$f_1(x_1,x_2)$", r"$f_2(x_1,x_2)$"])
plt.grid(True)
plt.xlim([1, 4.5])
```

1.5.1 Newton method for systems of equations

To derive the Newton method for a system of equations, we proceed in an analogous way as in the case of a scalar equation. The (multivariate) first-order Taylor series of $\mathbf{f}$ around an initial point $\mathbf{x}^{(0)}$ (sufficiently close to $\boldsymbol{\alpha}$) gives:

$$\mathbf{0} = \mathbf{f}(\boldsymbol{\alpha}) = \mathbf{f}(\mathbf{x}^{(0)}) + J_{\mathbf{f}}(\mathbf{x}^{(0)})(\boldsymbol{\alpha} - \mathbf{x}^{(0)}) + \text{“small term”}. \tag{1.10}$$

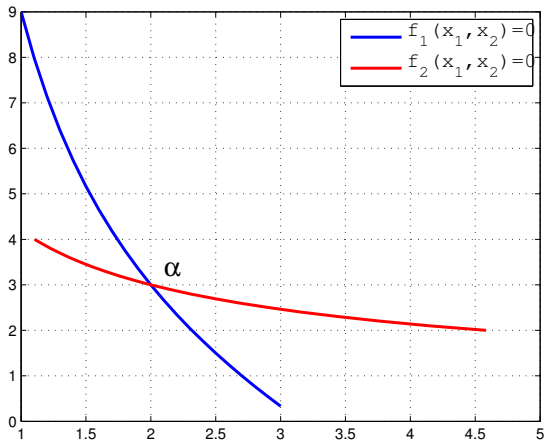


Figure 1.6: Intersection of the two functions $f_1(x_1, x_2) = 0$ (blue) and $f_2(x_1, x_2) = 0$ (red) from (1.9)

Here, the *Jacobian* matrix $J_{\mathbf{f}} \in \mathbb{R}^{n \times n}$ is defined by

$$J_{\mathbf{f}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \vdots \\ \vdots & & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \dots & & \frac{\partial f_n}{\partial x_n} \end{bmatrix}.$$

After neglecting the "small term", the equation (1.10) becomes a linear system in $\boldsymbol{\alpha}$. Solving this linear system, we find an approximation of the root $\boldsymbol{\alpha}$ which we will name $\mathbf{x}^{(1)}$ and for which

$$J_{\mathbf{f}}(\mathbf{x}^{(0)})(\mathbf{x}^{(1)} - \mathbf{x}^{(0)}) = -\mathbf{f}(\mathbf{x}^{(0)}).$$

The Newton method is obtained by repeating the process for $\mathbf{x}^{(1)}$. The solution of linear systems (needed to compute $\mathbf{x}^{(1)}$) will be discussed in Chapters 4 and 5. For the moment, we will simply use the `Python` command numpy.linalg.solve(A,b) to solve a linear system $A\mathbf{x} = \mathbf{b}$.

We can stop the iterations when $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \mathsf{tol}$ where we use $\|v\|$ to denote the Euclidean norm of a vector $v \in \mathbb{R}^n$: $\|v\| := \sqrt{v_1^2 + \ldots + v_n^2}$.

Algorithm 1.7: Newton method for systems of equations

Data: $\mathbf{f}$, $J_{\mathbf{f}}$, $\mathbf{x}^{(0)}$, tol, nmax
Result: $\boldsymbol{\alpha}$, res, niter
$r^{(0)} = \mathsf{tol} + 1$; $k = 0$;
while $r^{(k)} > \mathsf{tol}$ *AND* $k < nmax$ **do**
  Solve linear system $J_{\mathbf{f}}(\mathbf{x}^{(k)})\delta\mathbf{x} = -\mathbf{f}(\mathbf{x}^{(k)})$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta\mathbf{x}$;
  $r^{(k+1)} = \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|$;
  $k = k + 1$;
end
$\boldsymbol{\alpha} = \mathbf{x}^{(k)}$, res=$r^{(k)}$, niter=$k$;

The convergence analysis is done in the same way as for the scalar case. Here, we will just quote the main result.

Theorem 1.4. *Consider a function* $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ *of class* C^2 *and a root* $\boldsymbol{\alpha}$ *of* $\mathbf{f}$. ***If*** $\det(J_{\mathbf{f}}(\alpha)) \neq 0$ *(the Jacobian matrix is invertible)* ***then*** *the Newton method converges with order (at least) 2 for all* $\mathbf{x}^{(0)}$ *sufficiently close to* $\boldsymbol{\alpha}$.

Example 1.4. *Let us apply the Newton method to the system* (1.9) *with the initial point* $\mathbf{x}^{(0)} = (1, 0)$. *The first iteration of the Newton method is computed as follows:*

$$\mathbf{f}(\mathbf{x}^{(0)}) = \begin{bmatrix} f_1(1,0) \\ f_2(1,0) \end{bmatrix} = \begin{bmatrix} -9 \\ -57 \end{bmatrix},$$

$$J_{\mathbf{f}}(\mathbf{x}) = \begin{bmatrix} 2x_1 + x_2 & x_1 \\ 3x_2^2 & 1 + 6x_1x_2 \end{bmatrix} \qquad \Longrightarrow \qquad J_{\mathbf{f}}(\mathbf{x}^{(0)}) = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}.$$

Hence, we have to solve the linear system

$$\begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \delta x_1 \\ \delta x_2 \end{bmatrix} = \begin{bmatrix} 9 \\ 57 \end{bmatrix},$$

which has the solution $\delta\mathbf{x} = (-24, 57)$. *This gives* $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \delta\mathbf{x} = (-23, 57)$.

Example 1.5. *We now solve* (1.9) *using the Newton method in* **Python**, *with the help of the function* newtonsys *available on Moodle. We choose* $\mathbf{x}^{(0)} = [3, 4]^\top$ *as initial value.*

```
import numpy as np
from functions import newtonsys

f = lambda x: np.array([x[0] ** 2 + x[0] * x[1] - 10,
                        x[1] + 3 * x[0] * x[1] ** 2 - 57])
df = lambda x: np.array([[2 * x[0] + x[1], x[0]],
```

```
                          [3 * x[1] ** 2,1 + 6 * x[0] * x[1]]])
x0 = [3, 4]
x, inc, niter = newtonsys(f, df, x0, 1e-8, 1000)
print("x= " + str(x[-1]))
print("number of iterations " + str(niter))
print("increments=" + str(inc))

# OUTPUT
# x= [2. 3.]
# number of iterations 5
# increments=[1.11487660e+00 3.27957413e-01 3.58120064e-02
#  1.96157782e-04 1.10946122e-09]
```

We observe that the method converges to the exact solution $\boldsymbol{\alpha} = (2, 3)$ *in only 5 iterations. We can also check the order of convergence. As in the scalar case, the function* newtonsys *gives us also the list of the increments* $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|$ *in the variable* inc*. This means we can compute the ratio of two consecutive increments*

$$\frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|}{\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|^p}$$

for different values of p*.*

```
import numpy as np
from functions import newtonsys

f = lambda x: np.array([x[0] ** 2 + x[0] * x[1] - 10,
                        [1] + 3 * x[0] * x[1] ** 2 - 57])
df = lambda x: np.array([[2 * x[0] + x[1], x[0]],
                         [3 * x[1] ** 2, 1 + 6 * x[0] * x[1]]])
x0 = [3, 4]
x, inc, niter = newtonsys(f, df, x0, 1e-8, 1000)
print("x= " + str(x[-1]))
print("number of iterations " + str(niter))
print("increments=" + str(inc))
print("ratio with p=1")
print(inc[1:] / inc[:-1])
print("ratio with p=2")
print(inc[1:] / inc[:-1] ** 2.0)
print("ratio with p=3")
print(inc[1:] / inc[:-1] ** 3.0)

# OUTPUT
# ratio with p=1
# [2.94164765e-01 1.09197124e-01 5.47743064e-03 5.65596310e-06]
# ratio with p=2
# [0.2638541  0.33296129 0.15294956 0.02883374]
# ratio with p=3
# [  0.23666664   1.01525771   4.27090173 146.99260448]
```

For $p = 2$, the ratio appears to be constant, allowing us to conclude that the method converges with order 2.

We conclude this section with a few words on ensuring global convergence of the Newton method, that is, convergence also from initial points that are not necessarily close to a root. For this purpose, let us first write the Newton method in compact form:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \big(J_{\mathbf{f}}(\mathbf{x}^{(k)})\big)^{-1}\mathbf{f}(\mathbf{x}^{(k)}).$$

Divergence of the Newton method is often caused by "overshooting", that is, the new approximation $\mathbf{x}^{(k+1)}$ gets further away from the root than $\mathbf{x}^{(k)}$. Overshooting can be cured by considering the damped Newton method:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \lambda_k\big(J_{\mathbf{f}}(\mathbf{x}^{(k)})\big)^{-1}\mathbf{f}(\mathbf{x}^{(k)}),$$

where $0 < \lambda_k \leq 1$ is a damping factor. This damping factor is selected from the set $\{1, 1/2, 1, 4, \ldots\}$ by choosing the first value for which $\mathbf{f}(\mathbf{x}^{(k+1)})$ becomes (significantly) smaller than $\mathbf{f}(\mathbf{x}^{(k)})$.

Chapter 2

Curve fitting

Suppose that we have at our disposal measures of a quantity y, for example the water temperature of the Leman lake, at n different depths. Let x_i, $i = 1, \ldots, n$ be the depths at which we collect the measures and y_i, $i = 1, \ldots, n$ the corresponding measured temperatures.

We try to find a continuous function $p(x)$ that best describes the data (x_i, y_i), namely $p(x_i) \approx y_i$.

A different way of framing this problem is the following: We make the hypothesis that a function f exists which describes the link between y and x, that is, $y = f(x)$. Importantly, f is defined for all x and not only for the values x_i at which we obtained the measures y_i. We say that $y = f(x)$ is a *conceptual model*. For the example above, we imagine that there exists a function $y = f(x)$ that gives the temperature of the water at every depth of the lake. However, our only knowledge about this function are the values (x_i, y_i) that we measured. We are aiming at building the complete function $f(x)$ from the available measures. This will allow us to estimate the water temperature at every depth and not only at the measured ones x_i.

Example 2.1 (Dynamic viscosity of water)**.** *The viscosity of water depends on the temperature, as shown in Table 2.1 and Figure 2.1 (left). Suppose that we are interested in the viscosity of water at 25°C. This temperature is not in the table. To find a good approximation of the viscosity at this temperature, we can proceed as follows: First find a continuous function* $p(T)$ *that* interpolates *the data, that is,*

$$\textit{find } p(T) \qquad \textit{such that} \qquad p(T_i) = \nu_i, \tag{2.1}$$

where $T_1 = 10, T_2 = 20, \ldots, T_{10} = 100$ *are the temperature values from the table and* $\nu_1 = 1.308, \nu_2 = 1.002, \ldots, \nu_{10} = 0.2822$ *are the corresponding viscosity values. A problem of the form* (2.1) *is called* interpolation problem.

Once such a function p *is found, we can evaluate the viscosity at a temperature of 25 °C using* $\nu = p(25)$. *Figure 2.1 (right) shows a pos-*

Temperature [°C]	Viscosity [mPa · s]
10	1.308
20	1.002
30	0.7978
40	0.6531
50	0.5471
60	0.4658
70	0.4044
80	0.3550
90	0.3150
100	0.2822

Table 2.1: Viscosity of water depending on the temperature (source: Wikipedia)

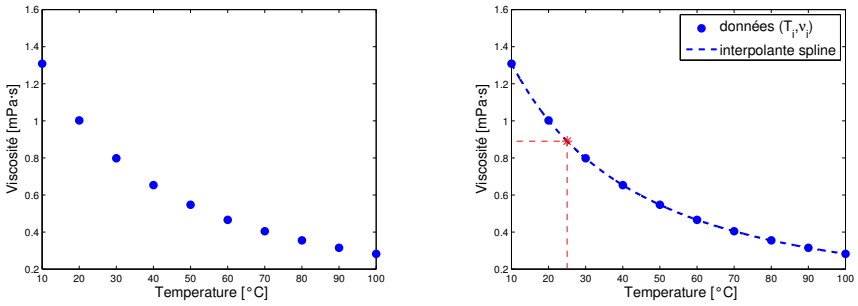


Figure 2.1: Left: Available date of water viscosity vs. temperature. Right: Spline function interpolating the data. In red: Viscosity at 25°C

sible interpolating function obtained by spline interpolation*, which will be discussed in Section 2.3.*

Example 2.2 (Dog bone tensile test)**.** *We want to measure the mechanical characteristics of a material. To this end, we perform tensile testing on a sample with the geometry shown in Figure 2.2 (dog bone). The sample is submitted to a stress σ (force by area). The corresponding strain ε is measured by a device called* strain gauge*. We apply multiple stresses σ_i, $i = 1, \ldots, n$ uniformly distributed (also called equidistant) between 0 and $\sigma_{\max}$ and we measure the corresponding strains ε_i. We then want to characterize*

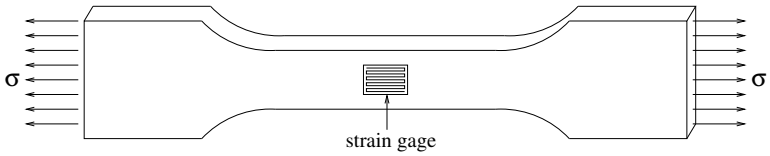


Figure 2.2: Sample of a material for tensile testing

the stress-strain law of this material. However, the values measured for ε_i *are affected by non-negligible measurement error. Figure 2.3 (left) shows possible data obtained in such an experiment. Let us imagine that there exists a true function* f *that links the strain to the stress:* $\varepsilon = f(\sigma)$ *and that the measurements* ε_i *are given by*

$$\varepsilon_i = f(\sigma_i) + \eta_i,$$

where η_i *contains measurement errors.*

We aim at obtaining a good approximation $p(\sigma)$ *for the true function* $f(\sigma)$ *from the data* $(\sigma_i, \varepsilon_i)$*. In this case, it is* not *a good idea to find an* interpolating *function, as such a function would also interpolate the measurement error. It would be better to aim at finding a function* $p(\sigma)$ *that approximates the data well (but not perfectly) and at the same time filters the measurement error. For this purpose, one often searches for* p *in a class of functions* $\mathcal{W}$ *(for example fixed-degree polynomials) that minimizes the sum of the squared data distances:*

$$p = \underset{q \in \mathcal{W}}{\operatorname{argmin}} \sum_{i=1}^{n} |\varepsilon_i - q(\sigma_i)|^2.$$

This is called least-squares approximation*. Figure 2.3 (middle) shows a reconstruction of the stress-strain law by spline interpolation of the data, while Figure 2.3 (right) shows the reconstruction obtained using least-squares polynomial approximation of third degree. Intuitively, the least-squares approximation appears to more sensible.*

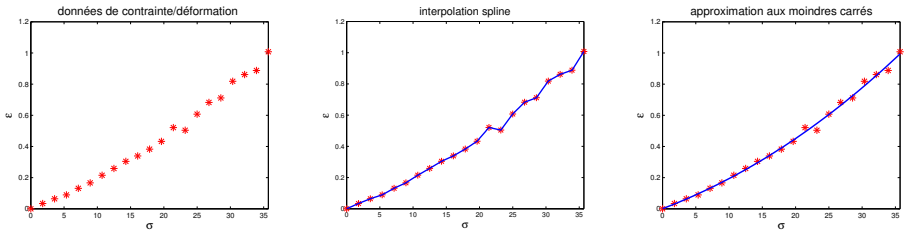


Figure 2.3: Left: Strain measures obtained with a strain gauge. Middle: Spline interpolation of data. Right: Least-squares approximation with polynomial of degree three.

2.1 Polynomial interpolation of data

We first consider the case where measures are not affected by measurement errors and the goal is to interpolate data. Suppose we have $n + 1$ data (x_i, y_i), with $i = 1, \ldots, n + 1$, and that there is a function f (unknown to

us) that links the variables y and x. Therefore, our model is

$$y_i = f(x_i), \qquad i = 1, \ldots, n+1.$$

A common approach is to *find a polynomial of degree (at most) n that interpolates the data.*

Definition 2.1. *Consider given data (x_i, y_i), $i = 1, \ldots, n+1$. A **polynomial interpolating** this data is a polynomial p_n of degree at most n, such that*

$$y_i = p_n(x_i), \qquad i = 1, \ldots, n+1.$$

We will see below, in Section 2.1, that the polynomial from Definition 2.1 always exists and is unique, provided that the values of x_i are mutually distinct. Indeed, for $n = 1$, it is simple to see that there is exactly one line (first degree polynomial) that passes through the two points (x_1, y_1) and (x_2, y_2). Likewise, there is exactly one parabola (second degree polynomial) that interpolates three points (x_1, y_1), (x_2, y_2), and (x_3, y_3).

Construction using Vandermonde matrix

The most common representation of a polynomial of degree at most n is in terms of the monomial basis $(1, x, \ldots, x^n)$:

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n, \qquad a_0, \ldots, a_n \in \mathbb{R}.$$

Now, the task is to find the $n+1$ unknown coefficients $a_0, \ldots, a_n$ from the data (x_i, y_i), $i = 1, \ldots, n+1$, by using the $n+1$ interpolation conditions $y_i = p_n(x_i)$:

$$\begin{cases} p_n(x_1) = a_0 + a_1 x_1 + a_2 x_1^2 + \ldots + a_n x_1^n = y_1, \\ p_n(x_2) = a_0 + a_1 x_2 + a_2 x_2^2 + \ldots + a_n x_2^n = y_2, \\ \vdots \\ p_n(x_{n+1}) = a_0 + a_1 x_{n+1} + a_2 x_{n+1}^2 + \ldots + a_n x_{n+1}^n = y_{n+1}. \end{cases} \tag{2.2}$$

This is a linear system of $n+1$ equations in the $n+1$ unknowns $a_0, \ldots, a_n$. It can be written in matrix form as follows:

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n+1} & x_{n+1}^2 & \cdots & x_{n+1}^n \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{bmatrix}}_{\mathbf{y}}. \tag{2.3}$$

The V matrix is called a *Vandermonde matrix*. Using the fact that the coefficients $a_0, \ldots, a_n$ are uniquely determined (because p_n is uniquely determined), it follows that the linear system (2.3) has a unique solution and,

hence, the matrix V is invertible (provided that the values of x_i are mutually distinct).

However, the matrix V is infamous for becoming a very *ill-conditioned* matrix as n increases; we will discuss this notion in more detail in Chapter 4. This makes the *numerical* solution of (2.3) difficult because small errors on the right-hand side (e.g., rounding errors due to the floating point number representation) are amplified in the solution and can lead to numerical solutions of very poor quality. We have to be careful when using this method for larger n (already $n = 15$ can be problematic).

Example 2.3 (Dynamic viscosity of water)**.** *Let us consider again the example of water viscosity. We want to find the polynomial of degree* 9 *that interpolates the 10 measures from Table 2.1. The Vandermonde method explained above is utilized by the* **Python** *command* p_coef=numpy.polyfit(x,y,n)*, where* x *is the vector containing the temperatures,* y *is the vector containing the measures (water viscosity) and* p_coef *is the vector containing the coefficients of the degree* n *interpolating polynomial in* decreasing *order:* p_coef=$(a_n, a_{n-1}, \ldots, a_0)$.

```
import numpy as np

T = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
nu = [1.308, 1.002, 0.7978, 0.6531, 0.5471, 0.4658, 0.4044,
      0.3550, 0.3150, 0.2822]
p_coef = np.polyfit(T, nu, 9)
print("p_coef:", p_coef)

# OUTPUT
# p_coef: [-3.76432981e-16  1.86061508e-13 -3.93806217e-11
#           4.66770833e-09 -3.40309606e-07  1.58054062e-05
#          -4.71143269e-04  9.11859623e-03 -1.30803159e-01
#           2.04700000e+00]
# p_coef: [-3.76432981e-16 1.86061508e-13 -3.93806217e-11
```

To plot the obtained polynomial, we can use the command np.polyval .

```
import matplotlib.pyplot as plt

T_fine = np.arange(10, 100.1, 0.1)  # fine grid for visualization
p = np.polyval(p_coef, T_fine)  # evaluation of polynomial in T_fine
plt.plot(T_fine, p, "b")
plt.plot(T, nu, "r*")
```

Figure 2.4 shows the graph of the interpolating polynomial. For this application, the polynomial appears to describe the behavior of the data very well.

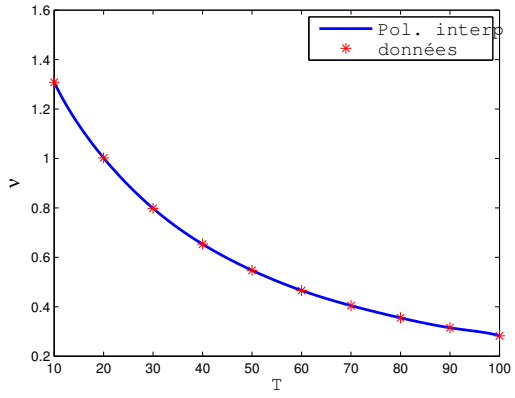


Figure 2.4: Data of water viscosity depending on temperature (Table 2.1) and degree 9 interpolating polynomial.

Construction using Lagrange basis

The Lagrange basis of polynomials offers an attractive alternative to the Vandermonde method for constructing interpolating polynomials. This alternative does not require the solution of any linear system and also tends to be numerically more reliable for larger values of n. We start by defining the Lagrange basis.

Definition 2.2. *Given mutually distinct nodes* $x_1, x_2, \dots x_{n+1} \in \mathbb{R}$ ***the corresponding basis of Lagrange polynomials*** *are the* $n+1$ *polynomials*

$$\begin{aligned}\phi_i(x) &= \frac{(x-x_1)(x-x_2)\dots(x-x_{i-1})(x-x_{i+1})\dots(x-x_{n+1})}{(x_i-x_1)(x_i-x_2)\dots(x_i-x_{i-1})(x_i-x_{i+1})\dots(x_i-x_{n+1})} \\ &= \prod_{\substack{j=1 \\ j\neq i}}^{n+1} \frac{(x-x_j)}{(x_i-x_j)}\end{aligned}$$

for $i = 1, \dots, n+1$.

It can be verified that $(\phi_1, \dots, \phi_{n+1})$ represents a basis of the vector space of polynomials of degree at most n. By their definition, these polynomials have the following important property:

$$\phi_i(x_k) = 0 \text{ if } k \neq i \text{ and } \phi_i(x_i) = 1. \tag{2.4}$$

In other words, the polynomial $\phi_i(x)$ equals 0 at every node x_k except for the node x_i where it equals 1. Figure 2.5 shows the polynomials ϕ_1, ϕ_3 and ϕ_5 associated with the nodes $x_1 = 0, x_2 = 0.2, x_3 = 0.4, \dots, x_6 = 1$.

Thanks to (2.4), the construction of an interpolating polynomial becomes very simple in the basis of Lagrange polynomials.

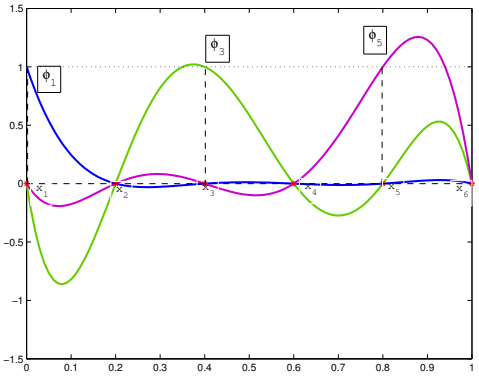


Figure 2.5: Lagrange polynomials ϕ_1, ϕ_3, ϕ_5 associated with the nodes $x_1 = 0, x_2 = 0.2, x_3 = 0.4, \ldots, x_6 = 1$.

Proposition 2.1. *The polynomial p_n interpolating the data (x_i, y_i), with $i = 1, \ldots, n+1$, is given by*

$$p_n(x) = \sum_{i=1}^{n+1} y_i \phi_i(x). \tag{2.5}$$

Because each ϕ_i has degree n, it follows that p_n has degree at most n. Moreover, the interpolating property follows from (2.4):

$$p_n(x_k) = y_1 \underbrace{\phi_1(x_k)}_{=0} + y_2 \underbrace{\phi_2(x_k)}_{=0} + \ldots + y_k \underbrace{\phi_k(x_k)}_{=1} + \ldots + y_{n+1} \underbrace{\phi_{n+1}(x_k)}_{=0} = y_k.$$

Proposition 2.1 also shows that the polynomial interpolation problem from Definition 2.1 always admits a solution. Moreover, the basis property of $\phi_1, \ldots, \phi_{n+1}$ implies that this is the only solution.

Example 2.4. *We want to build an interpolating polynomial for the data $(x_1 = 0, y_1 = 1)$ and $(x_2 = 1, y_2 = 2)$. Let us first start by building the Lagrange's basis*

$$\phi_1(x) = \frac{(x - x_2)}{(x_1 - x_2)} = \frac{x - 1}{-1} = 1 - x,$$
$$\phi_2(x) = \frac{(x - x_1)}{(x_2 - x_1)} = \frac{x - 0}{1} = x.$$

Finally, the interpolating polynomial is

$$p_1(x) = y_1\phi_1(x) + y_2\phi_2(x) = 1(1 - x) + 2x = x + 1$$

which can be easily verified.

For large n, the evaluation of p_n at a value x using directly the formula (2.5) becomes somewhat expensive because the number of operations needed grows quadratically with n. Let us mention barycentric interpolation formulas as a common approach to reduce this cost.

Error analysis

When discussing the quality and error of polynomial interpolation, it is important to pose the right questions. For example, one could ask whether the interpolating polynomial p_n is a good approximation of the "true" (but unknown) function f that generated the measurements y_i. However, this question is not very fruitful because there are infinitely many functions f that take the same values y_i at the nodes x_i and which could be arbitrarily far from p_n at points x outside the interpolation nodes. It is more meaningful to ask anther question: if we interpolate more and more measurements of f, will we obtain an increasingly accurate approximation p_n to f? In particular, if the number of measurements n approaches infinity, will the interpolating polynomial p_n converge to the "true" function f?

These questions appear theoretical but they are of high practical relevance. If p_n does not approach f as $n \to \infty$, then our reconstruction method is not correct in the limit. For larger (but finite) n one can then expect to see strange behavior, that p_n is not a good match to f. This will not happen when $p_n \to f$ when $n \to \infty$, but it is also important to know at which speed p_n converges to f. The faster p_n converges to f, the less measurements are needed to reconstruct f accurately.

The following example shows that the interpolating polynomials p_n do not always converge to the true function f.

Example 2.5. *Let us consider two functions*

$$f_1(x) = \sin(5x), \qquad f_2(x) = \frac{1}{1+(5x)^2},$$

defined in the interval $[-1, 1]$. *We use the measurements*

$$y_i = f_1(x_i), \qquad z_i = f_2(x_i), \quad i = 1, \ldots, n+1,$$
$$\{x_i\}_{i=1}^{n+1} \text{ equidistant nodes in the interval } [-1, 1].$$

Figure 2.6 shows the polynomials $p_n^{(1)}$ *and* $p_n^{(2)}$ *interpolating* $f_1(x_i)$ *and* $f_2(x_i)$, *respectively, for* $n = 4, 8, 16$. *For* $f_1(x) = \sin(5x)$, *the degree* 16 *interpolating polynomial is almost superimposed on the exact function and, hence,* $p_n^{(1)}$ *appears to converge to the exact function when increasing the number of measurements. On the other hand, for* $f_2(x) = 1/(1+(5x)^2)$, *the degree* 16 *interpolating polynomial provides a decent approximation of the true function close to interval center only, and a very poor approximation close to the interval endpoints (actually worse than the degree* 8 *interpolating polynomial).*

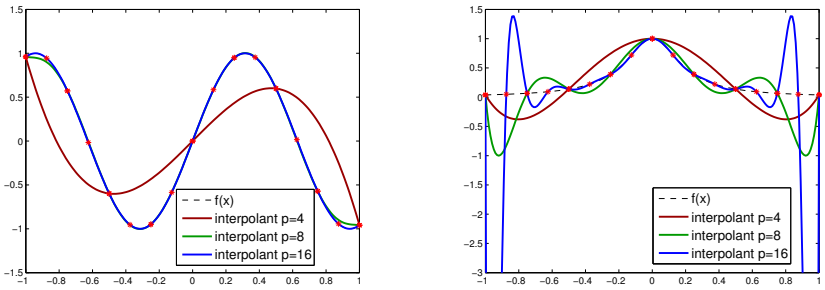


Figure 2.6: Interpolating polynomials $p_n^{(1)}$ (left) and $p_n^{(2)}$ (right) for $n = 4, 8, 16$.

Increasing the polynomial degree further, this effect becomes even stronger and the interpolating polynomial diverges *from the function f_2 when $|x|$ is higher than approximately* 0.6. *This is known as* Runge's phenomenon.

The following result provides an upper bound on the interpolation error in terms of the higher-order derivatives of f.

Theorem 2.2. *Consider a function $f : [a, b] \to \mathbb{R}$ of class C^{n+1} and equidistant nodes $a = x_1 < x_2 < \ldots < x_{n+1} = b$ in $[a, b]$. Then the polynomial p_n of degree at most n that interpolates the data $(x_i, f(x_i))$, with $i = 1, \ldots, n+1$, satisfies the error bound*

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \frac{1}{4(n+1)} \left(\frac{b-a}{n}\right)^{n+1} \max_{x\in[a,b]} |f^{(n+1)}(x)|. \qquad (2.6)$$

Theorem 2.2 is difficult to prove; see, e.g., [Quarteroni, Sacco, Saleri, "Numerical Mathematics", Springer]. The result implies convergence if $|f^{(n+1)}(x)|$ grows more slowly on the interval $[a, b]$ than $\frac{4(n+1)n^{n+1}}{(b-a)^{n+1}}$ as $n \to \infty$. This is the case for the function $f_1(x) = \sin(5x)$ from Example 2.5, for which the derivatives satisfy

$$\max_{x\in[-1,1]} |f_1^{(n+1)}(x)| \leq 5^{n+1}$$

and, hence,

$$\max_{x\in[a,b]} |f(x)-p_n(x)| \leq \frac{2^{n+1}}{4(n+1)n^{n+1}} \max_{x\in[-1,1]} |f_1^{(n+1)}(x)| \leq \frac{10^{n+1}}{4(n+1)n^{n+1}} \xrightarrow{n\to\infty} 0.$$

On the other hand, for the function f_2 from Example 2.5, the derivatives grow much more quickly (one can show that $\max_{x\in[-1,1]} |f_2^{(n)}(x)| \sim n!5^n$) and convergence is not guaranteed by Theorem 2.2 and, in fact, convergence does not happen.

It is important to emphasize that the divergence of polynomial interpolation, even for very nice functions, is linked to the use of equidistant nodes

in the interval $[a, b]$. There are other choices, which are typically denser around the endpoints of the interval, that yield convergent interpolations if f is at least differentiable. This is the case when *Clenshaw-Curtis nodes* are used.

Definition 2.3. *The n* ***Clenshaw-Curtis nodes*** *on the interval $[a, b]$ are defined by*

$$x_i = \frac{a+b}{2} - \frac{b-a}{2} \cos\left(\frac{\pi(i-1)}{n}\right), \qquad i = 1, \ldots, n+1.$$

The Clenshaw-Curtis nodes are obtained through projecting uniformly distributed nodes on the semi-circle of center $(a+b)/2$ and radius $(b-a)/2$ on the horizontal axis; see Figure 2.7. Another good choice are Chebyshev

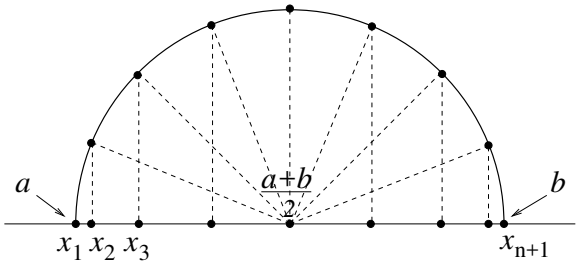


Figure 2.7: Clenshaw-Curtis nodes.

nodes. Yet, in the context of data interpolation, we usually do not have the luxury of choosing the position of the nodes x_i by ourselves. In most practical situations, the nodes are determined by the measurements; well before any interpolation is performed.

Stability of polynomial interpolation

In this section, we study the effects of data error on the quality of polynomial interpolation.

Suppose we want to determine the polynomial $p_n(x)$ that interpolates the data (x_i, y_i), $i = 1, \ldots, n+1$, with $a = x_1 < x_2 < \ldots < x_{n+1} = b$ and $y_i = f(x_i)$ for some function f.

Because of, e.g., measurement error, roundoff error, etc. the data is inevitably affected by error. Instead of y_i, we are actually collecting perturbed data $\tilde{y}_i = f(x_i) + \epsilon_i$ with an error ϵ_i that is assumed to be small: $|\epsilon_i| \leq \epsilon$ for $i = 1, \ldots, n+1$.

Letting $\tilde{p}_n$ denote the interpolating polynomial for the perturbed data $(x_i, \tilde{y}_i)$, we now want to estimate the distance between the "true" interpolating polynomial p_n and $\tilde{p}_n$. For this purpose, we represent both polynomials

in the Lagrange basis with respect to $x_1, \ldots, x_{n+1}$:

$$p_n(x) = \sum_{i=1}^{n+1} y_i \phi_i(x), \qquad \tilde{p}_n(x) = \sum_{i=1}^{n+1} (y_i + \epsilon_i)\phi_i(x).$$

Taking the difference and using the triangle inequality, we obtain that

$$|\tilde{p}_n(x) - p_n(x)| = \left| \sum_{i=1}^{n+1} \phi_i(x)\epsilon_i \right| \leq \sum_{i=1}^{n+1} |\phi_i(x)||\epsilon_i| \leq \left(\sum_{i=1}^{n+1} |\phi_i(x)| \right) \epsilon$$

for every $x \in [a, b]$.

Definition 2.4. *The **Lebesgue constant** associated with nodes $x_1, \ldots, x_{n+1} \in [a, b]$ is defined as*

$$L_n = \sup_{x \in [a,b]} \sum_{i=1}^{n+1} |\phi_i(x)|.$$

In summary, we have the following *stability result*:

$$\max_{x \in [a,b]} |\tilde{p}(x) - p(x)| \leq L_n \cdot \epsilon, \qquad \epsilon = \max_{i=1,\ldots,n+1} |\epsilon_i|, \tag{2.7}$$

that is, the input error ϵ is potentially magnified by L_n. In particular, when the Lebesgue constant L_n is small, small input errors $|\epsilon_i| \leq \epsilon$ lead to small perturbations in the interpolating polynomial. In this case, we say that the polynomial interpolation on the nodes $x_1, \ldots, x_{n+1}$ is *stable* or *well conditioned*. If, on the contrary, the Lebesgue constant is very large then the polynomial interpolation on the nodes $x_1, \ldots, x_{n+1}$ is *badly conditioned*; it has bad stability properties.

For polynomial interpolation on equidistant or Clenshaw-Curtis nodes, one has the following results:

$$\textit{equidistant nodes} \qquad L_n \sim \frac{2^{n+1}}{en \log n}, \quad n \to \infty, \tag{2.8}$$

$$\textit{Clenshaw-Curtis nodes} \qquad L_n \sim \frac{2}{\pi} \log n, \quad n \to \infty. \tag{2.9}$$

This tells us that polynomial interpolation on Clenshaw-Curtis nodes is relatively well conditioned (stable) as the Lebesgue constant grows only very slowly with n. On the other hand, polynomial interpolation on equidistant nodes is very badly conditioned as the Lebesgue constant grows exponentially with n, which means that small perturbations on data are potentially highly amplified.

Example 2.6. *Let us consider the function $f(x) = \sin(5x)$ from Example 2.5, for which we have already seen that polynomial interpolation provides good results.*

Given distinct nodes $\{x_i\}_{i=1}^{n+1}$ *on the interval* $[-1,1]$ *and the exact evaluations* $y_i = f(x_i)$*, we consider perturbed evaluations*

$$\tilde{y}_i = f(x_i) + \epsilon_i,$$

where $\epsilon_i \in [-10^{-2}, 10^{-2}]$ *are randomly generated errors of magnitude* $|\epsilon_i| \leq 10^{-2}$*. The polynomials* p_n *and* $\tilde{p}_n$ *interpolating* (x_i, y_i) *and* $(x_i, \tilde{y}_i)$*, respectively, are computed using* `Python` *.*

Figure 2.8 shows the two interpolating polynomials for equidistant nodes and Clenshaw-Curtis nodes. In the first case, it is clear that the maximum distance between the two polynomials is of order 1*, about 100 times larger than the perturbations* ϵ_i*. On the other hand, in the second case, the distance between the polynomials is very small and, in fact, we cannot even distinguish the two polynomials in the figure.*

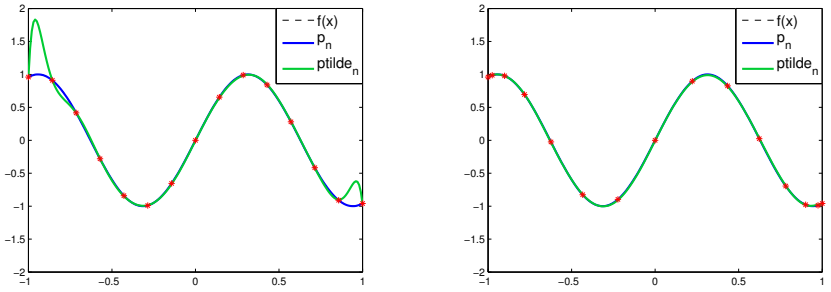


Figure 2.8: Example 2.6: Polynomials $p_n, \tilde{p}_n$ interpolating (x_i, y_i), $(x_i, \tilde{y}_i)$ perturbed by the random errors $|\epsilon_i| \leq 10^{-2}$. Left: Equidistant nodes; right: Clenshaw-Curtis nodes.

2.2 Piecewise linear interpolation

We have seen in the previous section that enlarging the degree n of the interpolating polynomial can be an excellent idea, especially when the nodes can be freely chosen, but can also lead to disastrous results. Instead of increasing the degree, another idea is to subdivide the interval $[a, b]$ and apply polynomial interpolation to each sub-interval. When degree 1 polynomials are used, this idea leads to *piecewise linear interpolation*.

To formalize this idea, let us consider equidistant nodes $a = x_1 < x_2 < \ldots < x_{n+1} = b$ nodes in the interval $[a, b]$ and $y_i = f(x_i)$ the corresponding measurements, which come from the evaluation of an unknown function f. We set $I_i = [x_i, x_{i+1}]$, which is an interval of length $h = (b-a)/n$.

Definition 2.5. ***A piecewise linear polynomial*** *interpolating the data* (x_i, y_i)*, with* $i = 1, \ldots, n+1$*, is a function* $p_{1,h}$ *such that*

- $p_{1,h}$ *is degree 1 polynomial in every interval* I_i, $i = 1, \ldots, n$,
- $p_{1,h}(x_i) = y_i$, $i = 1, \ldots, n+1$.

Definition 2.5 requires $p_{1,h}$ to be a degree 1 polynomial on an interval I_i. Taking into account that $p_{1,h}$ interpolates the data (x_i, y_i) and (x_{i+1}, y_{i+1}), the Lagrange basis approach allows us to express $p_{1,h}$ restricted to I_i in the form

$$p_{1,h}(x) = y_i \frac{x - x_{i+1}}{x_i - x_{i+1}} + y_{i+1} \frac{x - x_i}{x_{i+1} - x_i}, \quad \forall x \in I_i.$$

The `Python` command p1h=numpy.interp(x,y,x_fine) performs piecewise linear interpolation, where the vector x contains the nodes, the vector y contains the measurements vector, x_fine is a point or a vector of points where we want to evaluate $p_{1,h}$, and the output p1h contains the evaluations $p_{1,h}$(x_fine).

Example 2.7. *Once again, we use the function* $f_2(x) = 1/(1 + (5x)^2)$ *from Example 2.5, for which we have observed that high-degree polynomial interpolation is problematic. The following* `Python` *code computes the piecewise linear interpolation on 9 equidistant nodes in the interval* $[-1, 1]$*:*

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

f = lambda x: 1.0 / (1 + (5 * x) ** 2)
n = 8
x = np.linspace(-1, 1, n + 1)  # interpolation nodes
y = f(x)
# measures
xfine = np.linspace(-1, 1, 201)  # fine grid
p1h = np.interp(xfine, x, y)  # evaluation of p1h on fine grid
plt.plot(xfine, p1h, "b")
plt.plot(xfine, f(xfine), "k--")
plt.legend(["p1h, n=8", "f(x)", "data"])
```

Figure 2.9 shows the obtained result. We notice that $p_{1,h}$ *does not exhibit oscillations and gives a decent (although not quite excellent) approximation of* f_2.

Error analysis

The error analysis of piecewise linear interpolation is a direct consequence of Theorem 2.2. In fact, in every interval I_i, we are computing a linear interpolation of the data (x_i, y_i) and (x_{i+1}, y_{i+1}). This means we can apply the result of Theorem 2.2 with $n = 1$ and $b - a = x_{i+1} - x_i = h$:

$$\max_{x \in I_i} |f(x) - p_{1,h}(x)| \leq \frac{h^2}{8} \max_{x \in I_i} |f^{(2)}(x)|.$$

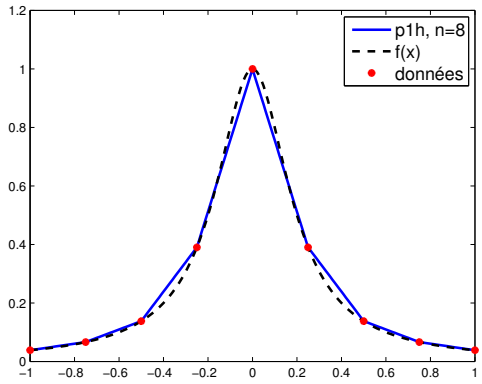


Figure 2.9: Piecewise linear interpolation of the function $f(x) = 1/(1+(5x)^2)$ on 9 equidistant nodes in the interval $[-1, 1]$.

Therefore,

$$\max_{x\in[a,b]} |f(x) - p_{1,h}(x)| = \max_{i=1,\ldots,n} \max_{x\in I_i} |f(x) - p_{1,h}(x)| \leq \frac{h^2}{8} \max_{x\in[a,b]} |f^{(2)}(x)|.$$

The following theorem summarizes this result.

Theorem 2.3. *Consider a function $f : [a, b] \to \mathbb{R}$ of class C^2 and equidistant nodes $a = x_1 < x_2 < \ldots < x_{n+1} = b$. Then the piecewise linear polynomial $p_{1,h}$ interpolating the data $(x_i, f(x_i))$ satisfies the error bound*

$$\max_{x\in[a,b]} |f(x) - p_{1,h}(x)| \leq Ch^2 \max_{x\in[a,b]} |f^{(2)}(x)|, \tag{2.10}$$

with $h = (b-a)/n$ and $C = 1/8$.

For an error bound of the form (2.10), we say that convergence (with respect to h) is *quadratic* or of *second order*. More generally, a convergence of order $q \in \mathbb{N}$ requires that the approximation error of a piecewise defined approximation is bounded by a constant (independent of h) times h^q.

To verify numerically that piecewise linear approximation has second order, one needs to check that the error is asymptotically proportional to h^2. In other words, if we double the number of points (and therefore half the length of every sub-interval), the error will be roughly divided by 4. The following `Python` snippet implements this idea for the function $f_2(x) = 1/(1+(5x)^2)$ from Example 2.5 by using $n = 16, 32, 64, 128, 256$ sub-intervals and estimating the error by computing the error on a fine grid.

```
import numpy as np

f = lambda x: 1.0 / (1 + (5 * x) ** 2)
xfine = np.linspace(-1, 1, 201)  # fine grid
```

```
err = np.array([])
h = np.array([])
for n in 2 ** np.arange(4, 9):
    h = np.append(h, 2.0 / n)
    x = np.linspace(-1, 1, n + 1)  # interpolation nodes
    y = f(x)  # measures
    p1h = np.interp(xfine, x, y)  # evaluation of p1h on fine grid
    err = np.append(err, max(abs(f(xfine) - p1h)))
print("h: ", h)
print("err: ", err)

# OUTPUT
# h:  [0.125     0.0625    0.03125   0.015625  0.0078125]
# err:  [0.05353602 0.02069974 0.00535163 0.00138879 0.00035451]
```

Indeed, the output indicates that the error is divided by 4 (approximately) when the number of sub-intervals is doubled. To observe this behavior more clearly, we first note that the error can be written as

$$\mathrm{err}_h = \max_{x\in[a,b]} |f(x) - p_{1,h}(x)| \sim Ch^2$$

with $C = \max_{x\in[a,b]} |f^{(2)}(x)|/8$. Applying the logarithm in base 10 (or any other base) to both sides, we obtain that

$$\log_{10}(\mathrm{err}_h) \sim \log_{10}(Ch^2) = \log_{10}(C) + 2\log_{10}(h).$$

Therefore $\log_{10}(\mathrm{err}_h)$ *is a linear function of* $\log_{10}(h)$ with slope 2, which corresponds to the convergence order. This means the order can be observed from the graph of $\log_{10}(\mathrm{err}_h)$ with respect to $\log_{10}(h)$. In `Python`, there is no need to take logarithms explicitly – the command loglog(x,y) from matplotlib.pyplot, applies logarithmic scaling to both axes.

```
import matplotlib.pyplot as plt

plt.loglog(h, err, "b")
plt.loglog(h, h, "k--")
plt.loglog(h, h**2, "k-.")
plt.grid(True)
plt.legend(["err_h", "slope 1", "slope 2"])
```

Figure 2.10 shows the obtained result. On the same graph we also plotted the curves $y = h$ and $y = h^2$. In loglog scaling, these two curves match two lines of slope 1 and 2 respectively.

Stability of piecewise linear interpolation

Let us consider again the effect of input errors (such as measurement errors) on the interpolation. The perturbed data is $\tilde{y}_i = f(x_i) + \epsilon_i$, with a (small)

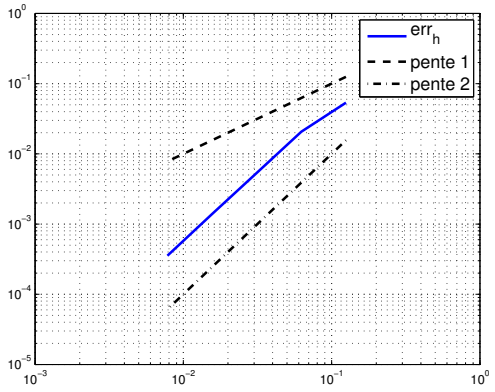


Figure 2.10: Graph in logarithmic scale of the piecewise linear interpolation error for $f(x) = 1/(1+(5x)^2)$ on $[-1,1]$ versus $h = 2/n$ (length of each sub-interval).

error ϵ_i satisfying $|\epsilon_i| \leq \epsilon$ for $i = 1, \ldots, n+1$. Let $p_{1,h}$ and $\tilde{p}_{1,h}$ denote the piecewise linear polynomials interpolating the non-perturbed data (x_i, y_i) and the perturbed data $(x_i, \tilde{y}_i)$, respectively. On each sub-interval $I_i = [x_i, x_{i+1}]$ we then have

$$p_{1,h}(x) = y_i \frac{x - x_{i+1}}{x_i - x_{i+1}} + y_{i+1} \frac{x - x_i}{x_{i+1} - x_i}$$

and

$$\tilde{p}_{1,h}(x) = (y_i + \epsilon_i) \frac{x - x_{i+1}}{x_i - x_{i+1}} + (y_{i+1} + \epsilon_{i+1}) \frac{x - x_i}{x_{i+1} - x_i}.$$

Taking the difference and applying the triangle inequality, it holds for every $x \in I_i$ that

$$|\tilde{p}_{1,h}(x) - p_{1,h}(x)| \leq |\epsilon_i| \left| \frac{x - x_{i+1}}{x_i - x_{i+1}} \right| + |\epsilon_{i+1}| \left| \frac{x - x_i}{x_{i+1} - x_i} \right| \leq \epsilon.$$

For the last inequality, we used the relation

$$\left| \frac{x - x_{i+1}}{x_i - x_{i+1}} \right| + \left| \frac{x - x_i}{x_{i+1} - x_i} \right| = \frac{x_{i+1} - x}{x_{i+1} - x_i} + \frac{x - x_i}{x_{i+1} - x_i} = 1 \quad \forall x \in I_i.$$

Finally, taking the maximum in every sub-interval, we get

$$\max_{x \in [a,b]} |\tilde{p}_{1,h}(x) - p_{1,h}(x)| = \max_{i=1,\ldots,n} \max_{x \in I_i} |\tilde{p}_{1,h}(x) - p_{1,h}(x)| \leq \epsilon. \tag{2.11}$$

We can conclude that piecewise linear interpolation is perfectly stable! Small measurement errors induce equally small perturbations in the piecewise linear interpolation.

Example 2.8. *As in Examples 2.5 and 2.6, we consider* $f(x) = \sin(5x)$. *Let* $\{x_i\}_{i=1}^{n+1}$ *denote equidistant nodes in the interval* $[-1, 1]$, $y_i = f(x_i)$, *and*

$$\tilde{y}_i = f(x_i) + \epsilon_i,$$

where $\epsilon_i \in [-10^{-2}, 10^{-2}]$ *are randomly generated errors of magnitude* $|\eta_i| \leq 10^{-2}$. *Using* `Python`, *we compute the piecewise linear polynomials* $p_{1,h}$, $\tilde{p}_{1,h}$ *interpolating* (x_i, y_i), $(x_i, \tilde{y}_i)$. *The obtained results are plotted in Figure 2.11. The distance between* $p_{1,h}$ *and* $\tilde{p}_{1,h}$ *is very small; in fact, it is visually indistinguishable.*

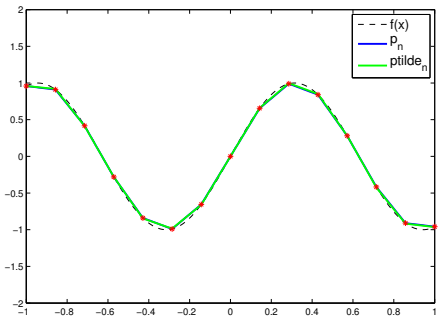


Figure 2.11: Example 2.8: Piecewise linear polynomials $p_{1,h}$ and $\tilde{p}_{1,h}$ interpolating exact data(x_i, y_i) and perturbed data $(x_i, \tilde{y}_i)$ with $|\epsilon_i| \leq 10^{-2}$.

2.3 Spline interpolation

Although piecewise linear interpolation is convergent when $n \to \infty$ and has favorable stability properties, it has two major drawbacks: (1) The convergence is only of second order. (2) The piecewise linear polynomial is continuous but not even differentiable (at the nodes). The second drawback is of particular relevance in graphical applications such as CAD (Computer Aided Design), where "smoother" curves are needed to, e.g., represent curved edges or surfaces. In these cases, one often aims at ensuring a continuous second derivative. *Splines* are the standard tool for constructing piecewise functions with favorable smoothness properties. In the following, we will focus on cubic splines.

Definition 2.6. *Consider data* (x_i, y_i), $i = 1, \ldots, n+1$ *with* $a = x_1 < x_2 < \ldots < x_{n+1} = b$, *and let let* $I_i = [x_i, x_{i+1}]$ *for* $i = 1, \ldots, n$. *An* interpolating cubic spline *is a function* $s_{3,h}$ *such as*

- $s_{3,h} \in C^2([a, b])$,

- $s_{3,h}$ *is a cubic polynomial on every interval* I_i,
- $s_{3,h}$ *interpolates the data:* $s_{3,h}(x_i) = y_i$ *for* $i = 1, \dots, n+1$.

Definition 2.6 makes sense for arbitrary (mutually distinct) nodes; but we will only consider equidistant nodes in the following, in which case $h = (b-a)/n$ denotes the length of each sub-interval.

The fact that $s_{3,h}$ is of class C^2 implies that not only the function $s_{3,h}$ but also its first and second derivatives are continuous functions. Because $s_{3,h}$ is a polynomial on each interval I_i, it is infinitely often differentiable in the *interior* $]x_i, x_{i+1}[$. In turn, the condition $s_{3,h} \in C^2([a,b])$ amounts to checking smoothness at the nodes x_i for $i = 2, \dots, n$:

$$s_{3,h}(x_i^-) = s_{3,h}(x_i^+), \quad i = 2, \dots, n, \tag{2.12}$$
$$s'_{3,h}(x_i^-) = s'_{3,h}(x_i^+), \quad i = 2, \dots, n, \tag{2.13}$$
$$s''_{3,h}(x_i^-) = s''_{3,h}(x_i^+), \quad i = 2, \dots, n. \tag{2.14}$$

Here, $g(x^-)$ and $g(x^+)$ are used to denote the left-sided and right-sided limits of a function at g at x.

On every sub-interval I_i, the spline is a cubic polynomial, which can be written as

$$s_{3,h}\big|_{I_i} = a_i + b_i x + c_i x^2 + d_i x^3, \qquad i = 1, \dots, n,$$

with the coefficients (a_i, b_i, c_i, d_i) to be determined. Consequently, there are $4n$ unknown coefficients (4 in each sub-interval). We also have $3(n-1)$ continuity conditions in the internal nodes and $n+1$ interpolating conditions $s_{3,h}(x_i) = y_i$.

unknowns	$4n$
continuity conditions	$3(n-1)$
interpolating conditions	$n+1$
degrees of freedom	$4n - 3(n-1) - (n+1) = 2$

Thus, the number of degrees of freedom *nearly* match the number of unknowns. The two additional degrees of freedom can be chosen to impose two additional conditions on the spline. Four popular choices are:

- *natural spline*: $s''_{3,h}(x_1) = 0$, $s''_{3,h}(x_{n+1}) = 0$,
- *prescribed slope at endpoints*: $s'_{3,h}(x_1) = \alpha_1$, $s'_{3,h}(x_{n+1}) = \alpha_2$ for given $\alpha_1, \alpha_2 \in \mathbb{R}$,
- *not-a-knot condition* (default in `Python`): continuity of the third derivative in x_2 and x_n

$$s'''_{3,h}(x_2^-) = s'''_{3,h}(x_2^+), \qquad s'''_{3,h}(x_n^-) = s'''_{3,h}(x_n^+),$$

- *periodic spline*: $s'_{3,h}(x_1^-) = s'_{3,h}(x_{n+1}^+)$, $s''_{3,h}(x_1^-) = s''_{3,h}(x_{n+1}^+)$, usually assuming $y_1 = y_{n+1}$.

Example 2.9. *Let us consider the following data:* $(x_1, y_1) = (-1, -1)$, $(x_2, y_2) = (0, 1)$, $(x_3, y_3) = (1, 1)$. *We look for the natural spline interpolating the data.*

The general expression of the spline is

$$\begin{aligned} s_{3,h}(x) &= a_1 + b_1 x + c_1 x^2 + d_1 x^3 && \text{for } x \in [-1, 0] \\ s_{3,h}(x) &= a_2 + b_2 x + c_2 x^2 + d_2 x^3 && \text{for } x \in [0, 1]. \end{aligned}$$

Continuity conditions:

$$\begin{aligned} s_{3,h}(x_2^-) = s_{3,h}(x_2^+), &\quad \Longrightarrow \quad a_1 = a_2 \\ s'_{3,h}(x_2^-) = s'_{3,h}(x_2^+), &\quad \Longrightarrow \quad b_1 = b_2 \\ s''_{3,h}(x_2^-) = s''_{3,h}(x_2^+), &\quad \Longrightarrow \quad c_1 = c_2. \end{aligned}$$

Natural spline condition

$$\begin{aligned} s''_{3,h}(x_1) = 0, &\quad \Longrightarrow \quad 2c_1 - 6d_1 = 0 \\ s''_{3,h}(x_3) = 0, &\quad \Longrightarrow \quad 2c_2 + 6d_2 = 0. \end{aligned}$$

We can then simplify the general expression as follows:

$$\begin{aligned} s_{3,h}(x) &= a + bx + cx^2 + \frac{1}{3}cx^3 && \text{for } x \in [-1, 0] \\ s_{3,h}(x) &= a + bx + cx^2 - \frac{1}{3}cx^3 && \text{for } x \in [0, 1]. \end{aligned}$$

Finally, let us impose the 3 interpolating conditions

$$\begin{aligned} s_{3,h}(-1) = -1, &\quad \Longrightarrow \quad a - b + \frac{2}{3}c = -1, \\ s_{3,h}(0) = 1, &\quad \Longrightarrow \quad a = 1, \\ s_{3,h}(1) = 1, &\quad \Longrightarrow \quad a + b + \frac{2}{3}c = 1, \end{aligned}$$

and we find $a = 1$, $b = 1$ *and* $c = -\frac{3}{2}$. *Finally, the cubic interpolating spline is*

$$s_{3,h}\big|_{[-1,0]} = 1 + x - \frac{3}{2}x^2 - \frac{1}{2}x^3, \qquad s_{3,h}\big|_{[0,1]} = 1 + x - \frac{3}{2}x^2 + \frac{1}{2}x^3.$$

The general construction of a cubic spline is quite similar to what has been done in Example 2.9. In the general case, the determination of the coefficients requires the solution of a (tridiagonal) linear system, which can be solved efficiently using the techniques discussed in Section 4.5.

2.3.1 Error analysis

The error of cubic spline interpolation satisfies the following bounds.

Theorem 2.4. *Consider a function $f : [a, b] \to \mathbb{R}$ of class C^4 and equidistant nodes $a = x_1 < x_2 < \ldots < x_{n+1} = b$. Then any cubic spline $s_{3,h}$ interpolating the data $(x_i, f(x_i))$ satisfies*

$$\max_{x\in[a,b]} |f(x) - s_{3,h}(x)| \leq C_0 h^4 \max_{x\in[a,b]} |f^{(4)}(x)|, \tag{2.15}$$

$$\max_{x\in[a,b]} |f'(x) - s'_{3,h}(x)| \leq C_1 h^3 \max_{x\in[a,b]} |f^{(4)}(x)|, \tag{2.16}$$

$$\max_{x\in[a,b]} |f''(x) - s''_{3,h}(x)| \leq C_2 h^2 \max_{x\in[a,b]} |f^{(4)}(x)|, \tag{2.17}$$

where $h = (b-a)/n$ and C_0, C_1 and C_2 are constants not depending on h.

Theorem 2.4 tells us that cubic spline interpolation results in convergence order 4 for the interpolation error. Additionally, the first and second derivatives of the function f are approximated as well, with convergence order 3 and 2, respectively.

Example 2.10. *We apply cubic spline interpolation to the function $f_2(x) = 1/(1+(5x)^2)$ from Example 2.5 for an increasing number of sub-intervals: $n = 16, 32, 64, 128, 256$.*

```
import numpy as np
from scipy.interpolate import CubicSpline

f = lambda x: 1.0 / (1 + (5 * x) ** 2)
xfine = np.linspace(-1, 1, 201)  # fine grid
err = np.array([])
h = np.array([])
for n in 2 ** np.arange(4, 9):
    h = np.append(h, 2.0 / n)
    x = np.linspace(-1, 1, n + 1)  # interpolation nodes
    y = f(x)  # measurements
    s3h = CubicSpline(x, y)  # defines cubic spline s3h
    err = np.append(err, max(abs(f(xfine) - s3h(xfine))))

print("h: ", h)
print("err: ", err)

# OUTPUT
# h:  [0.125     0.0625    0.03125   0.015625  0.0078125]
# err:  [3.71093415e-03 6.37335632e-04 3.60663244e-05
#        2.05343200e-06 1.24035520e-07]
```

We observe that doubling the number of sub-intervals (i.e., h is halved), decreases the error decreases roughly by a factor 16. Visualizing the error with respect to h on a loglog plot clearly reveals order 4 because the error curve is nearly parallel to a line with slope 4; see Figure 2.12.

```
import matplotlib.pyplot as plt

plt.loglog(h, err, "b")
plt.loglog(h, h**4, "k--")
plt.grid(True)
plt.legend(["err_h", "slope 4"])
```

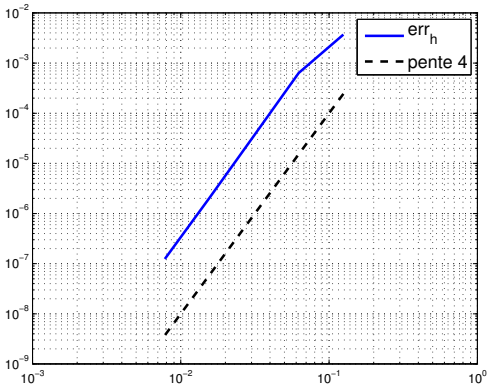


Figure 2.12: Loglog plot of spline interpolation error for $f(x) = 1/(1+(5x)^2)$ versus sub-interval length $h = 2/n$.

2.4 Least-squares approximation

When the data is perturbed anyway, it may not be meaningful to exactly interpolate it. Rather, it would be sufficient to approximate the data, with an approximation error preferably on the level of the measurement error. In this section, we will consider the specific case of polynomial least-squares approximation for this purpose, which includes the wildly popular[1] linear regression as a special case.

Suppose we have $n+1$ data (x_i, y_i), with $i = 1, \ldots, n+1$, and that the values y_i come from the evaluation of an unknown $f(x)$ function. Then our model for the perturbed data takes the form

$$y_i = f(x_i) + \varepsilon_i, \tag{2.18}$$

and we are aiming at approximating the unknown function $f(x)$ from this (perturbed) data. In the following, $\mathbb{P}_m$ denotes the vector space of polynomials of degree at most m.

[1] See `https://xkcd.com/1725/`.

Definition 2.7. *The degree m* ***least-squares polynomial*** *approximating given data (x_i, y_i), with $i = 1, \ldots, n+1$, is the polynomial satisfying*

$$p_m^{\mathsf{LS}} = \operatorname*{argmin}_{q \in \mathbb{P}_m} \sum_{i=1}^{n+1} (y_i - q(x_i))^2.$$

In other words, p_m^{LS} is such that

$$\sum_{i=1}^{n+1} (y_i - p_m^{\mathsf{LS}}(x_i))^2 \leq \sum_{i=1}^{n+1} (y_i - q(x_i))^2, \qquad \forall q \in \mathbb{P}_m.$$

The use of the (squared) Euclidean norm for measuring the error in Definition 2.7 has a strong statistical motivation. Suppose that the errors ε_i are random variables, independent and identically distributed (iid) with expected value $\mathbb{E}[\varepsilon_i] = 0$ and variance $\mathbb{V}\mathrm{ar}[\varepsilon_i] = \sigma^2$. For example, this is the case if ε_i are normally distributed, $\varepsilon_i \sim N(0, \sigma^2)$. Then p_m^{LS} represents the maximum likelihood estimate (MLE), that is, p_m^{LS} is the most likely choice of polynomial given the observations $y_1, \ldots, y_{n+1}$.

Figure 2.13 shows a set of 21 measures and the linear least-squares fit (*regression line*) to the data.

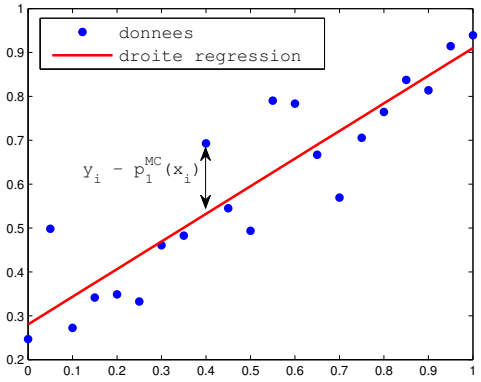


Figure 2.13: Regression line (in red) of the 21 measures (in blue).

Computation of least-squares polynomial for $m = 1$

In the case of linear regression ($m = 1$) one can work out a simple formula for the coefficients of the least-squares fit. For a linear functions of the form

$$q(x) = a_0 + a_1 x, \qquad a_0, a_1 \in \mathbb{R},$$

we are trying to solve

$$\min_{q\in\mathbb{P}_1}\sum_{i=1}^{n+1}(y_i-q(x_i))^2=\min_{(a_0,a_1)\in\mathbb{R}^2}\sum_{i=1}^{n+1}(y_i-(a_0+a_1x_i))^2.$$

In other words, we are looking for the minimum of the function

$$\phi(a_0,a_1)=\sum_{i=1}^{n+1}(y_i-(a_0+a_1x_i))^2.$$

This function is differentiable and, hence, any minimum has to satisfy the conditions

$$\frac{\partial\phi}{\partial a_0}=0,\qquad \frac{\partial\phi}{\partial a_1}=0,$$

which match

$$\begin{cases}\dfrac{\partial\phi}{\partial a_0}=-2\displaystyle\sum_{i=1}^{n+1}(y_i-a_0-a_1x_i)=0\\[2ex] \dfrac{\partial\phi}{\partial a_1}=-2\displaystyle\sum_{i=1}^{n+1}x_i(y_i-a_0-a_1x_i)=0\end{cases}$$

$$\Longrightarrow\begin{cases}(n+1)a_0+\left(\sum_{i=1}^{n+1}x_i\right)a_1=\sum_{i=1}^{n+1}y_i\\ \left(\sum_{i=1}^{n+1}x_i\right)a_0+\left(\sum_{i=1}^{n+1}x_i^2\right)a_1=\sum_{i=1}^{n+1}x_iy_i.\end{cases}$$

The latter is a linear system of two equations in the two unknowns (a_0,a_1):

$$\underbrace{\begin{bmatrix}\sum_{i=1}^{n+1}1 & \sum_{i=1}^{n+1}x_i\\ \sum_{i=1}^{n+1}x_i & \sum_{i=1}^{n+1}x_i^2\end{bmatrix}}_{A}\underbrace{\begin{bmatrix}a_0\\a_1\end{bmatrix}}_{\mathbf{a}}=\underbrace{\begin{bmatrix}\sum_{i=1}^{n+1}y_i\\ \sum_{i=1}^{n+1}x_iy_i\end{bmatrix}}_{\mathbf{b}}.\tag{2.19}$$

Unless all x_i are identical, the matrix A is invertible and the solution of this linear system is unique. This solution necessarily minimizes $\phi(a_0,a_1)$ and thus yields the coefficients of the regression line.

There is another way to arrive at the linear system (2.19). Let us introduce the Vandermonde matrix $V\in\mathbb{R}^{(n+1)\times 2}$ corresponding to the monomials 1 and x estimated at the points x_i, $i=1,\dots,n+1$:

$$V=\begin{bmatrix}1 & x_1\\ 1 & x_2\\ \vdots & \vdots\\ & \\ 1 & x_{n+1}\end{bmatrix}$$

Defining the vector $\mathbf{y}=(y_1,y_2,\dots,y_{n+1})^\top$, we have

$$A=V^\top V,\qquad \mathbf{b}=V^\top\mathbf{y}.$$

Hence, the system (2.19) can be written in the more compact form

$$V^\top V\mathbf{a} = V^\top \mathbf{y}.$$

Computation of least-squares polynomial (general case)

In the general case of a degree m polynomial, we define the Vandermonde matrix $V \in \mathbb{R}^{(n+1)\times(m+1)}$ corresponding to the monomials $1, x, \ldots, x^m$ estimated at the points x_i, $i = 1, \ldots, n+1$:

$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & & & \vdots \\ & & & & \\ 1 & x_{n+1} & x_{n+1}^2 & \ldots & x_{n+1}^m \end{bmatrix}.$$

Given a polynomial $q(x) = a_0 + a_1 x + \ldots + a_m x^m$, this allows us to write

$$\begin{bmatrix} q(x_1) \\ q(x_2) \\ \vdots \\ q(x_{n+1}) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & & & \vdots \\ 1 & x_{n+1} & x_{n+1}^2 & \ldots & x_{n+1}^m \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix} = V\mathbf{a}$$

and

$$\sum_{i=1}^{n+1} (y_i - q(x_i))^2 = \|\mathbf{y} - V\mathbf{a}\|^2. \tag{2.20}$$

Performing a least-squares fit of the the data (x_i, y_i) with a degree m polynomial is the same as finding the vector of coefficients $\mathbf{a} = (a_0, a_1, \ldots, a_m)^\top$ that minimizes the quantity (2.20):

$$\min_{q \in \mathbb{P}_m} \sum_{i=1}^{n+1} (y_i - q(x_i))^2 \qquad \Leftrightarrow \qquad \min_{\mathbf{a} \in \mathbb{R}^{m+1}} \|\mathbf{y} - V\mathbf{a}\|^2.$$

We have the following result, which generalizes the one we obtained for $m = 1$.

Proposition 2.5. *The polynomial $p_m^{\mathsf{LS}}(x) = a_0 + a_1 x + \ldots + a_m x^m$ is the least-squares polynomial approaching the data (x_i, y_i), $i = 1, \ldots, n+1$, if and only if $\mathbf{a} = (a_0, a_1, \ldots, a_m)^\top$ is solves the linear system*

$$V^\top V\mathbf{a} = V^\top \mathbf{y}. \tag{2.21}$$

The linear system (2.21) is usually called *normal equations*. It has a unique solution if and only if V has rank at least m. This is the case, for example, when $m \leq n$ and the points x_i are mutually distinct.

Remark 2.1. *For $m = n$, the Vandermonde matrix becomes a square matrix, which is invertible when the points x_i are mutually distinct. Therefore,*

$$V^\top V\mathbf{a} = V^\top \mathbf{y} \qquad \Leftrightarrow \qquad V\mathbf{a} = \mathbf{y}.$$

It follows that the coefficients vector corresponds to one of the polynomial interpolating *the data (x_i, y_i), $i = 1, \dots n+1$.*

In `Python`, the degree m least-squares polynomial is computed using the command pcoef=np.polyfit(x,y,m). For $m = n$, we obtain the interpolating polynomial.

Error analysis

Let us assume that $x_1 < x_2 < \cdots < x_{m+1}$. The analysis of the approximation error $E_m = \max_{x\in[x_1,x_{n+1}]} |f(x) - p_m^{\mathsf{LS}}(x)|$ is quite complicated. We will therefore limit ourselves to a few considerations.

Suppose that the "true" function $f(x)$ is a polynomial of degree $m \ll n$, that is, $f(x) = q_m(x) \in \mathbb{P}_m$. If there are no measurement errors then $y_i = q_m(x_i)$ for $i = 1, \dots, n+1$ and the degree m least-squares polynomial approximation recovers the function q_m: $p_m^{\mathsf{LS}}(x) = q_m(x)$. In fact, the polynomial $p_m^{\mathsf{LS}} = q_m(x)$ is the only degree m polynomial for which the sum of the squared differences is zero

$$\sum_{i=1}^{n+1}(y_i - p_m^{\mathsf{LS}}(x_i))^2 = \sum_{i=1}^{n+1}(q_m(x_i) - p_m^{\mathsf{LS}}(x_i))^2 = 0.$$

In the presence measurement errors ε_i, it is usually not possible to achieve zero error. If the errors ε_i are iid random variables with expected value zero and variance $\sigma^2 > 0$, one can prove that $E_m = \max_{x\in[x_1,x_{n+1}]} |q_m(x) - p_m^{\mathsf{LS}}(x)|$ is of the order of $\sigma\sqrt{\frac{m+1}{n+1}}$. Thus, for $m \ll n$ (many observations) one recovers the "ground truth" q_m quite accurately with the least-squares polynomial. The variance of the variables ε_i is usually not known, but it can be estimated by

$$\hat{\sigma}^2 = \frac{1}{n-m}\sum_{i=1}^{n+1}(y_i - p_m^{\mathsf{LS}}(x_i))^2.$$

In the the general case, when the function f is not necessarily a polynomial (but sufficiently smooth), we expect the least-squares approximation to be more and more accurate as the polynomial degree increases. However:

- There will always be an error of the order of $\sigma\sqrt{\frac{m+1}{n+1}}$, due to measurement error.
- If m becomes too large, the least-squares polynomial suffers from the same instabilities we saw gets for interpolating polynomials.

stress	strain	stress	strain
1.7850	0.0292	19.6350	0.4454
3.5700	0.0609	21.4200	0.5043
5.3550	0.0950	23.2050	0.5122
7.1400	0.1327	24.9900	0.6111
8.9250	0.1449	26.7750	0.7277
10.7100	0.2062	28.5600	0.7392
12.4950	0.2692	30.3450	0.8010
14.2800	0.2823	32.1300	0.8329
16.0650	0.3613	33.9150	0.9302
17.8500	0.4014	35.7000	1.0116

Table 2.2: Data of stress and strain from the tensile test of Example 2.2.

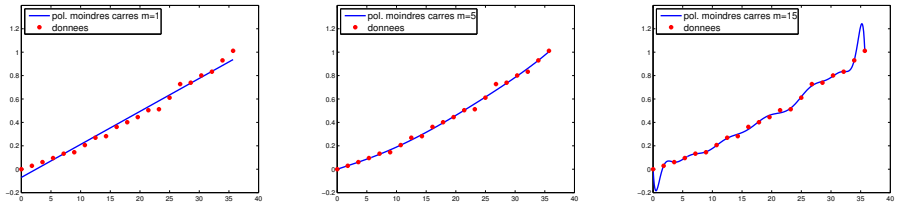


Figure 2.14: Approximation polynomials of the data from Table 2.2 to the least-squares of degree $m = 1, 5, 15$.

Example 2.11. *The Table 2.2 shows measurements for the tensile test from Example 2.2. Using* **Python**, *we compute the least-square polynomial approximation of degree* $m = 1, 5, 15$ *for the stress–strain relation. Assuming that the data is contained in the variables* x *and* y*, this is the code for* $m = 15$*:*

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

x = np.linspace(0, 35.7, 21)
y = [0, 0.0292, 0.0609, 0.0950, 0.1327, 0.1449, 0.2062, 0.2692,
     0.2823, 0.3613, 0.4014, 0.4454, 0.5043, 0.5122, 0.6111,
     0.7277, 0.7392, 0.8010, 0.8329, 0.9302, 1.0116]
xfine = np.arange(0, 35.8, 0.1)
pcoef = np.polyfit(x, y, 15)
p = np.polyval(pcoef, xfine)
plt.plot(xfine, p, "b-")
plt.plot(x, y, "r.", linewidth=2, markersize=15)
plt.legend(["pol. least squares m=15", "data"])
```

Figure 2.14 shows the obtained approximations. The degree 5 *polynomial seems to capture the behavior of the data better compared to the linear polynomial. The degree* $m = 15$ *polynomial shows very clear oscillations due to the fact that* m *is relatively large and close to* n*.*

Chapter 3

Numerical differentiation and integration

In this chapter, we aim at developing numerical methods to compute (approximately) the derivative of a function $f(x)$ at a point $x = \bar{x}$,

$$f'(\bar{x}),$$

as well as the integral of f over an interval $[a, b]$,

$$\int_a^b f(x)\,\mathrm{d}x.$$

These methods are very useful when f has a complicated expression and the exact (symbolic) computation of the derivative or the integral is complicated or even impossible.

3.1 Finite differences

Given a differentiable function $f : [a, b] \to \mathbb{R}$, we want to approximate numerically its derivative f' at a point $\bar{x}$ in the interval $[a, b]$, using only the evaluations of f at a few points in the interval. An immediate idea is to go back to the definition of derivatives from Analysis I and approximate the exact derivative $f'(\bar{x})$ by the *growth rate (slope)*:

$$f'(\bar{x}) \approx \frac{f(\bar{x} + h) - f(\bar{x})}{h} \tag{3.1}$$

for some $h > 0$ such that $\bar{x} + h \in [a, b]$. We expect the approximation to become increasingly accurate as h becomes smaller.

To analyze (3.1), we define the **forward finite differences formula**

$$\delta_h^+ f(\bar{x}) = \frac{f(\bar{x} + h) - f(\bar{x})}{h}. \tag{3.2}$$

Using the Taylor series of f around $\bar{x}$ gives

$$f(\bar{x}+h) = f(\bar{x}) + f'(\bar{x})h + \frac{f''(\xi)}{2}h^2, \quad \text{for some } \xi \in (\bar{x}, \bar{x}+h).$$

This implies

$$\delta_h^+ f(\bar{x}) = \frac{1}{h}\left[f(\bar{x}) + f'(\bar{x})h + \frac{f''(\xi)}{2}h^2 - f(\bar{x})\right] = f'(\bar{x}) + \frac{f''(\xi)}{2}h,$$

and therefore

$$|f'(\bar{x}) - \delta_h^+ f(\bar{x})| \leq \frac{1}{2}\max_{x\in[a,b]}|f''(x)|h,$$

which gives the following result:

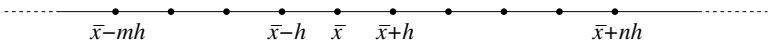
Lemma 3.1. *Consider a function $f : [a,b] \to \mathbb{R}$ of class C^2. Then the forward finite differences formula δ_h^+ satisfies the error bound*

$$|f'(\bar{x}) - \delta_h^+ f(\bar{x})| \leq Ch, \quad \forall \bar{x} \in [a, b-h],$$

with $C = \frac{1}{2}\max_{x\in[a,b]}|f''(x)|$.

As the error is is proportional to $h = h^1$, we say that the forward finite differences formula is of first order. Dividing h by 2, one expects the approximation error to be roughly divided by 2 as well.

The formula δ_h^+ uses only two function evaluations, $f(\bar{x})$ and $f(\bar{x}+h)$, to obtain an approximation of $f'(\bar{x})$. It is possible to construct finite differences formulas that use more function evaluations of f. In general, one considers the point $\bar{x}$ and n points on the right: $\bar{x}+h$, $\bar{x}+2h$, ..., $\bar{x}+nh$, and m points on the left: $\bar{x}-h$, $\bar{x}-2h$, ..., $\bar{x}-mh$:



We then make the following general definitions.

Definition 3.1. *A finite differences formula that uses $n+m+1$ points $\bar{x}+ih$, $i = -m, \ldots, n$, to approximate the derivative of f at $\bar{x}$ is an expression of the form*

$$D_h f(\bar{x}) = \frac{1}{h}\sum_{i=-m}^{n}\alpha_i f(\bar{x}+ih),$$

for some coefficients $\alpha_i \in \mathbb{R}$ not depending on h.

Definition 3.2. *A finite differences formula D_h is **consistent** if*

$$\lim_{h\to 0} D_h f(\bar{x}) = f'(\bar{x})$$

for sufficiently smooth f.

Definition 3.3. *A finite differences formula D_h is of* ***order*** **p** *if there is a constant C, which depends on f but* not *on h, such that*

$$|f'(\bar{x}) - D_h f(\bar{x})| \leq Ch^p,$$

for sufficiently smooth function f.

The forward finite differences formula (3.2) uses only two function evaluations, at $\bar{x}$ and $\bar{x} + h$, and is of first order. The following two popular formulas also use only function evaluations:

$$\textbf{backward finite differences:} \quad \delta_h^- f(\bar{x}) = \frac{f(\bar{x}) - f(\bar{x} - h)}{h}, \tag{3.3}$$

$$\textbf{central finite differences:} \quad \delta_h^c f(\bar{x}) = \frac{f(\bar{x} + h) - f(\bar{x} - h)}{2h}. \tag{3.4}$$

It is easily shown that the backward finite differences formula is also of first order. For central finite differences, however, we have that

$$\delta_h^c f(\bar{x}) = \frac{f(\bar{x}+h) - f(\bar{x}-h)}{2h} = \frac{1}{2h}\left[f(\bar{x}) + f'(\bar{x})h + \frac{f''(\bar{x})}{2}h^2 + \frac{f'''(\xi_1)}{6}h^3\right.$$
$$\left. -f(\bar{x}) + f'(\bar{x})h - \frac{f''(\bar{x})}{2}h^2 + \frac{f'''(\xi_2)}{6}h^3\right] = f'(\bar{x}) + \frac{h^2}{12}\left[f'''(\xi_1) + f'''(\xi_2)\right] \tag{3.5}$$

for some $\xi_1 \in (\bar{x}, \bar{x} + h)$ and $\xi_2 \in (\bar{x} - h, \bar{x})$. Therefore, this formula is of second order.

Lemma 3.2. *Consider a function $f : [a, b] \to \mathbb{R}$ of class C^3. Then the central finite differences formula δ_h^c satisfies the error bound*

$$|f'(\bar{x}) - \delta_h^c f(\bar{x})| \leq Ch^2, \quad \forall \bar{x} \in [a + h, b - h],$$

with $C = \frac{1}{6}\max_{x\in[a,b]} |f'''(x)|$.

Construction using interpolating polynomials

We start with an alternative way of deriving the forward finite differences formula: To approximate the derivative $f'(\bar{x})$ we first compute the line interpolating the points $(\bar{x}, f(\bar{x}))$ and $(\bar{x} + h, f(\bar{x} + h))$,

$$p_1(x) = f(\bar{x})\frac{\bar{x} + h - x}{h} + f(\bar{x} + h)\frac{x - \bar{x}}{h}.$$

Then we use the slope of this line as an approximation to the derivative,

$$p_1'(\bar{x}) = -\frac{1}{h}f(\bar{x}) + \frac{1}{h}f(\bar{x} + h) = \delta_h^+ f(\bar{x}),$$

which is seen to coincide with forward finite differences.

The procedure above can be generalized: Given the $m+n+1$ points $\bar{x}+ih$, $i=-m\ldots,n$, we first compute the degree $m+n$ polynomial p_{n+m} interpolating the function values $(\bar{x}+ih, f(\bar{x}+ih))$, $i=-m,\ldots,n$. Then the finite differences formula is obtained by computing the derivative of p_{n+m}:

$$D_h f(\bar{x}) = p'_{n+m}(\bar{x}).$$

Example 3.1. *We apply the procedure using the three points $\bar{x}-h$, $\bar{x}$ and $\bar{x}+h$. The quadratic polynomial p_2 interpolating $(\bar{x}+ih, f(\bar{x}+ih))$, $i=-1,0,1$ is given by*

$$p_2(x) = f(\bar{x}-h)\frac{(x-\bar{x})(x-\bar{x}-h)}{2h^2} + f(\bar{x})\frac{(x-\bar{x}+h)(x-\bar{x}-h)}{-h^2} + f(\bar{x}+h)\frac{(x-\bar{x}+h)(x-\bar{x})}{2h^2}. \quad (3.6)$$

Its derivative is

$$p'_2(x) = f(\bar{x}-h)\frac{2x-2\bar{x}-h}{2h^2} + f(\bar{x})\frac{2x-2\bar{x}}{-h^2} + f(\bar{x}+h)\frac{2x-2\bar{x}+h}{2h^2}$$

and the corresponding finite differences formula is

$$D_h f(\bar{x}) = p'_2(\bar{x}) = -\frac{f(\bar{x}-h)}{2h} + \frac{f(\bar{x}+h)}{2h},$$

which coincides with the central finite differences.

Construction using the indeterminate coefficients method

Another way to build finite differences is to start from a general expression

$$D_h f(\bar{x}) = \frac{1}{h}\sum_{i=-m}^{n} \alpha_i f(\bar{x}+ih)$$

and then look for coefficients α_i that lead to high order.

Example 3.2. *We consider again the three points $\bar{x}-h$, $\bar{x}$, $\bar{x}+h$ and the general formula*

$$D_h f(\bar{x}) = \frac{1}{h}\left[\alpha_{-1} f(\bar{x}-h) + \alpha_0 f(\bar{x}) + \alpha_1 f(\bar{x}+h)\right].$$

By the Taylor series, we have

$$\begin{aligned} D_h f(\bar{x}) = &\frac{\alpha_{-1}}{h}\left[f(\bar{x}) - f'(\bar{x})h + \frac{f''(\bar{x})}{2}h^2 + O(h^3)\right] + \\ &\frac{\alpha_0}{h} f(\bar{x}) + \\ &\frac{\alpha_1}{h}\left[f(\bar{x}) + f'(\bar{x})h + \frac{f''(\bar{x})}{2}h^2 + O(h^3)\right]. \end{aligned}$$

Therefore,

$$D_h f(\bar{x}) = \frac{1}{h}\underbrace{(\alpha_{-1}+\alpha_0+\alpha_1)}_{=0} f(\bar{x}) + \underbrace{(\alpha_1-\alpha_{-1})}_{=1} f'(\bar{x}) + \frac{h}{2}\underbrace{(\alpha_{-1}+\alpha_1)}_{=0} f''(\bar{x}) + O(h^2).$$

For the formula to yield a second order approximation of the first derivative, we have to impose the relations

$$\begin{cases} \alpha_{-1}+\alpha_0+\alpha_1 = 0 \\ \alpha_1-\alpha_{-1} = 1 \\ \alpha_1+\alpha_{-1} = 0 \end{cases}.$$

The unique solution is $\alpha_{-1} = -\frac{1}{2}$, $\alpha_0 = 0$, $\alpha_1 = \frac{1}{2}$, *which again corresponds to central finite differences.*

3.1.1 Higher-order derivatives

The two methods presented in the previous sections also allow us to construct finite differences formulas for approximating higher-order derivatives. Following up on Example 3.1, we can use the interpolating polynomial p_2 to obtain a finite differences formula that approximates the second derivative of f in $\bar{x}$:

$$D_h^2 f(\bar{x}) = p_2''(\bar{x}) = \frac{f(\bar{x}-h) - 2f(\bar{x}) + f(\bar{x}+h)}{h^2}.$$

Using Taylor series, one verifies that this formula is of second order.

3.1.2 Effects of round-off errors

When working on a computer (in finite-precision arithmetic), roundoff error will always put a limit on the accuracy we can possibly achieve. This effect is particularly visible when using finite difference formulas. To observe this, let us compute the derivative of the function $f(x) = \log(x)$ at $\bar{x} = 1$ with `Python` using forward finite differences. We use increasingly small values of h: $h = 10^{-1}, 10^{-2}, \ldots, 10^{-14}$.

```
import numpy as np

f = lambda x: np.log(x)
for i in range(1, 16):
    h = 10 ** (-i)
    dhf = (f(1 + h) - f(1)) / h
    print("h=%1.0e" % h, "   dhf=", dhf)

# OUTPUT
# h=1e-01     dhf= 0.9531017980432493
# h=1e-02     dhf= 0.9950330853168092
```

```
# h=1e-03    dhf= 0.9995003330834232
# h=1e-04    dhf= 0.9999500033329731
# h=1e-05    dhf= 0.9999950000398841
# h=1e-06    dhf= 0.9999994999180668
# h=1e-07    dhf= 0.9999999505838705
# h=1e-08    dhf= 0.9999999889225291
# h=1e-09    dhf= 1.0000000822240371
# h=1e-10    dhf= 1.000000082690371
# h=1e-11    dhf= 1.000000082735371
# h=1e-12    dhf= 1.000088900581841
# h=1e-13    dhf= 0.9992007221625909
# h=1e-14    dhf= 0.9992007221626359
# h=1e-15    dhf= 1.1102230246251559
```

The exact derivative is $f'(1) = 1$. One notices that the approximation error is as expected up to $h = 10^{-8}$. As we decrease h further, the approximation error first stagnates and then deteriorates. This is due to round-off error, which corrupts the computation of the difference $f(1 + h) - f(1)$. In the following, we will try to better understand this effect.

When evaluating the function $f(x)$, the computer usually makes some tiny round-off error. So, instead of $f(x)$, we actually compute a slightly corrupted function value

$$\hat{f}(x) = f(x)(1 + \eta_2)$$

for some η_2 of tiny magnitude. When working in double precision arithmetic (the default in `Python`), and f is well implemented, we expect $|\eta_2|$ to be of the order 10^{-16}. For simplicity, we assume $|\eta_2| \leq 10^{-16}$. Analogously, instead of $f(\bar{x} + h)$ the computer actually returns $f(\bar{x} + h)(1 + \eta_1)$ for some tiny η_1. These corruptions affect the finite difference formula as follows:

$$\begin{aligned}
\hat{\delta}_h^+ f(\bar{x}) &= \frac{\hat{f}(\bar{x} + h) - \hat{f}(\bar{x})}{h} = \frac{f(\bar{x} + h)(1 + \eta_1) - f(\bar{x})(1 + \eta_2)}{h} \\
&= \frac{f(\bar{x} + h) - f(\bar{x})}{h} + \frac{\eta_1}{h} f(\bar{x} + h) - \frac{\eta_2}{h} f(\bar{x}) \\
&= f'(\bar{x}) + \frac{f''(\xi)}{2} h + \frac{\eta_1}{h} f(\bar{x} + h) - \frac{\eta_2}{h} f(\bar{x})
\end{aligned}$$

where $|\eta_1|, |\eta_2| \leq 10^{-16}$ and $\xi \in (\bar{x}, \bar{x} + h)$. Finally, we have

$$|f'(\bar{x}) - \hat{\delta}_h^+ f(\bar{x})| \leq \max_{x \in [a,b]} |f''(x)| \frac{h}{2} + 2 \max_{x \in [a,b]} |f(x)| \frac{10^{-16}}{h}. \tag{3.7}$$

We notice that the first error term decreases proportionally with h (finite differences truncation error), while the second error term *grows* proportionally to $1/h$ (round-off errors). In particular, if h is too small, the second error term will dominate and accuracy will deteriorate.

What is the optimal value of h? Equation (3.7) gives us the error estimate

$$\varepsilon(h) \approx C_1 h + \frac{C_2 10^{-16}}{h},$$

with $C_1 = \frac{1}{2}\max_{x\in[a,b]} |f''(x)|$ and $C_2 = 2\max_{x\in[a,b]} |f(x)|$. The minimum of $\varepsilon(h)$ is computed by

$$\frac{\mathrm{d}\varepsilon}{\mathrm{d}h} = 0 \qquad \Longrightarrow \qquad C_1 - \frac{C_2 10^{-16}}{h^2} = 0,$$

which gives us (theoretically) the optimum value $h_{\mathsf{opt}} = \sqrt{C_2 10^{-16}/C_1}$. As C_1, C_2 are usually not available, a rule of thumb is to choose h_{opt} of the order $\sqrt{10^{-16}} = 10^{-8}$.

More generally, when considering a finite differences formula of order p, the error estimate takes the form

$$\varepsilon(h) \approx C_1 h^p + \frac{C_2 10^{-16}}{h},$$

which suggests an optimum value h_{opt} of order $\sqrt[p+1]{10^{-16}}$.

3.2 Numerical integration

Given $f : [a, b] \to \mathbb{R}$, we now aim at approximating numerically the integral

$$I = \int_a^b f(x)\,\mathrm{d}x,$$

using only evaluations of the function f at some points in the interval $[a, b]$. All popular methods take the form of a quadrature formula, as introduced by the following definition.

Definition 3.4. *Given points $a \le x_1 < x_2 < \ldots < x_n \le b$ (**quadrature nodes**) and scalars $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$ (**weights**), the **quadrature formula** $Q(f)$ for approximating the integral I takes the form*

$$Q(f) = \sum_{i=1}^{n} \alpha_i f(x_i).$$

Obviously, $Q(f)$ depends on the choice of quadrature nodes and weights, and we will now discuss several possible choices.

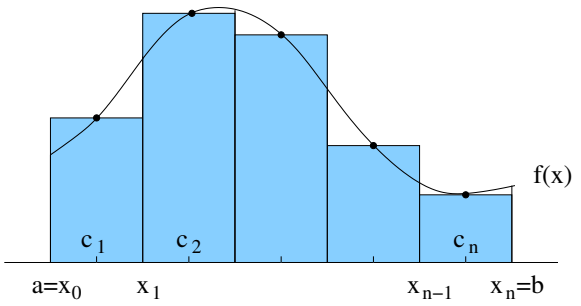


Figure 3.1: Graphical illustration of the composite midpoint formula.

Composite midpoint formula

One of the simplest ideas to approximate $I = \int_a^b f(x)dx$ is illustrated in Figure 3.1. The interval $[a, b]$ is divided into n sub-intervals $I_i = [x_{i-1}, x_i]$ of length $h = (b - a)/n$, that is, $x_i = a + ih$ for $i = 0, \ldots, n$. In every sub-interval I_i, the integral $\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x$ is approximated by the area of the rectangle with base h and height $f(\frac{x_{i-1}+x_i}{2})$. Letting $c_i = \frac{x_{i-1}+x_i}{2}$ denote the midpoint of I_i, this procedure corresponds to the quadrature formula

$$Q_h^{mp}(f) = \sum_{i=1}^{n} hf(c_i), \tag{3.8}$$

with quadrature nodes given by the n midpoints c_i, and the weights α_i equal to h. This formula is called *composite midpoint formula*. The term *composite* indicates that the quadrature formula is composed of applying the midpoint formula to every sub-interval I_i. If only one interval is considered ($i = 1$), the formula is called the (simple) midpoint formula.

The following `Python` function implements the composite midpoint formula:

```
import numpy as np

def midpoint(a, b, n, f):
    # Composite midpoint formula
    #  - a,b: boundaries of the integration interval
    #  - n: number of sub-intervals
    #  - f: function to integrate
    h = (b - a) / n
    xi = np.linspace(a + h / 2, b - h / 2, n)  # quadrature nodes
    alphai = np.full(n, h)  # weights
    Qh_mp = np.dot(alphai, f(xi))  # quadrature formula
    return Qh_mp
```

Composite trapezoidal formula

We continue dividing the interval $[a, b]$ in n sub-intervals $I_i = [x_{i-1}, x_i]$ of length $h = (b-a)/n$. However, instead of approximating $\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x$ by a rectangle, we now compute the area of the *trapezoid* defined by the four points $(x_{i-1}, 0)$, $(x_i, 0)$, $(x_i, f(x_i))$, $(x_{i-1}, f(x_{i-1}))$. This area is given by $\frac{h}{2}(f(x_{i-1}) + f(x_i))$, leading to the *composite trapezoid formula*

$$\begin{aligned} Q_h^{trap}(f) &= \sum_{i=1}^{n} \frac{h}{2}\left(f(x_{i-1}) + f(x_i)\right) \\ &= \frac{h}{2}f(x_0) + hf(x_1) + \ldots + hf(x_{n-1}) + \frac{h}{2}f(x_n). \end{aligned} \tag{3.9}$$

Figure 3.2 gives a graphical interpretation of the method. The composite

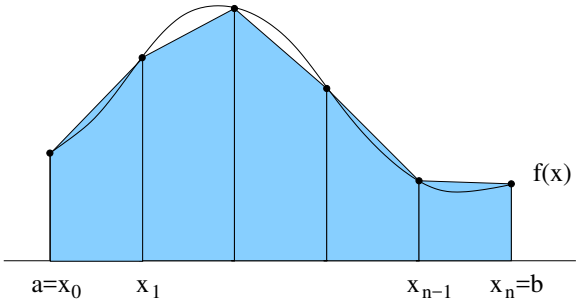


Figure 3.2: Graphical illustration of the composite trapezoidal formula.

trapezoidal formula uses the $n+1$ nodes x_i, $i = 0, \ldots, n$, and the weights $\alpha_i = h$, $i = 1, \ldots, n-1$, $\alpha_0 = \alpha_n = h/2$. This formula might appear more precise than the midpoint formula, but we will see that this is not necessarily true. Here is a possible implementation in `Python`:

```
import numpy as np

def trap(a, b, n, f):
    # Composite trapezoidal formula
    #  - a,b: boundaries of the integration interval
    #  - n: number of sub-intervals
    #  - f: function to integrate
    h = (b - a) / n
    xi = np.linspace(a, b, n + 1)  # quadrature nodes
    alphai = np.hstack((h / 2, np.full(n - 1, h), h / 2))  # weights
    Qh_trap = np.dot(alphai, f(xi))  # quadrature formula
    return Qh_trap
```

Composite Simpson formula

A third idea to build a quadrature formula is to use on every sub-interval the points x_{i-1}, x_i as well as the midpoint $c_i = \frac{x_{i-1}+x_i}{2}$. On every sub-

interval, we compute the quadratic polynomial $p_2^{(i)}$ interpolating the data $(x_{i-1}, f(x_{i-1}))$, $(x_i, f(x_i))$, $(c_i, f(c_i))$ and then approximate the integral $\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x$ by $\int_{x_{i-1}}^{x_i} p_2^{(i)}(x)\,\mathrm{d}x$. We leave the calculations as an exercise. The resulting formula is known as the *composite Simpson formula*:

$$\begin{aligned} Q_h^{simp}(f) &= \sum_{i=1}^{n} \frac{h}{6}\left(f(x_{i-1}) + 4f(c_i) + f(x_i)\right) \\ &= \frac{h}{6}f(x_0) + \frac{h}{3}\sum_{i=1}^{n-1} f(x_i) + \frac{2h}{3}\sum_{i=1}^{n} f(c_i) + \frac{h}{6}f(x_n), \end{aligned} \tag{3.10}$$

which uses the nodes x_i, $i = 0, \ldots, n$, and c_i, $i = 1, \ldots, n$ ($2n + 1$ nodes in total). Here is a possible `Python` implementation:

```
import numpy as np

def simpson(a, b, n, f):
    # Composite Simpson formula
    #  - a,b: boundaries of the integration interval
    #  - n: number of sub-intervals
    #  - f: function to integrate
    h = (b - a) / n
    xi = np.linspace(a, b, n + 1)
    # sub-interval boundaries
    alphai = (h / 3) * np.hstack((0.5, np.ones(n - 1), 0.5))
    # weights at x_i
    ci = np.linspace(a + h / 2, b - h / 2, n)
    # sub-interval mid-points
    betai = (2 * h / 3) * np.ones(n)
    # weights at c_i
    Qh_simp = np.dot(alphai, f(xi)) + np.dot(betai, f(ci))
    return Qh_simp
```

3.2.1 Error analysis

The degree of exactness of a quadrature formula is a qualitative measure for its accuracy.

Definition 3.5. *A quadrature formula $Q(f) = \sum_{i=1}^{n} \alpha_i f(x_i)$ has a **degree of exactness** r if it integrates every polynomial of degree at most r exactly, that is,*

$$Q(p) = \int_a^b p(x)\,\mathrm{d}x, \qquad \forall p \in \mathbb{P}_r, \tag{3.11}$$

but there are polynomials of degree $r + 1$ for which Q is not exact.

It suffices to check (3.11) for every monomial x^s, $s \leq r$:

$$Q(x^s) = \int_a^b x^s\,\mathrm{d}x, \qquad s = 0, 1, \ldots, r. \tag{3.12}$$

Indeed, by the linearity of integration and quadrature formulas, this implies for every polynomial $p(x) = \sum_{k=0}^{r} a_k x^k$ that

$$Q(p) = \sum_{i=1}^{n} \alpha_i p(x_i) = \sum_{i=1}^{n} \alpha_i \sum_{k=0}^{r} a_k x_i^k$$
$$= \sum_{k=0}^{r} a_k \underbrace{\sum_{i=1}^{n} \alpha_i x_i^k}_{=\int_a^b x^k dx} = \int_a^b \sum_{k=0}^{r} a_k x^k dx = \int_a^b p(x)dx,$$

and, hence, (3.11) is satisfied.

Let us emphasize that Definition 3.5 is intended for simple quadrature formulas, that is, the composite quadrature formula is constrained to a single sub-interval.

Example 3.3. *Let us determine the degree of exactness for the Simpson formula. We choose a sub-interval $I_i = [x_{i-1}, x_i]$ and check whether*

$$\frac{h}{6}\left[p(x_{i-1}) + 4p(c_i) + p(x_i)\right] = \int_{x_{i-1}}^{x_i} p(x)dx, \quad \textit{with} \quad p(x) = x^s$$

for $s = 0, 1, \ldots$. To simplify the calculations, we only consider the interval $I_i = [-1, 1]$, which has length $h = 2$ (but the result carries over to any other interval). We have

$$p(x) = 1, \qquad \frac{h}{6}(1 + 4 + 1) = 2 = \int_{-1}^{1} 1\,\mathrm{d}x$$
$$p(x) = x, \qquad \frac{h}{6}(-1 + 4 \cdot 0 + 1) = 0 = \int_{-1}^{1} x\,\mathrm{d}x$$
$$p(x) = x^2, \qquad \frac{h}{6}((-1)^2 + 4 \cdot (0)^2 + 1^2) = \frac{2}{3} = \int_{-1}^{1} x^2\,\mathrm{d}x$$
$$p(x) = x^3, \qquad \frac{h}{6}((-1)^3 + 4 \cdot (0)^3 + 1^3) = 0 = \int_{-1}^{1} x^3\,\mathrm{d}x.$$

On the other hand.

$$p(x) = x^4, \qquad \frac{h}{6}((-1)^4 + 4 \cdot (0)^4 + 1^4) = \frac{2}{3} \neq \int_{-1}^{1} x^4\,\mathrm{d}x = \frac{2}{5}.$$

Therefore, the Simpson formula has degree of exactness 3.

The midpoint and trapezoidal formulas turn out to have degree of exactness 1, which gives the following table:

	midpoint	trapezoidal	Simpson
exactness degree	1	1	3

We now quantify the accuracy of a composite quadrature formula.

Definition 3.6. *Consider a composite quadrature formula $Q_h(f)$ defined on n sub-intervals of length $h = (b-a)/n$ for approximating the integral $I = \int_a^b f(x)\,\mathrm{d}x$. One says that $Q_h(f)$ is of* ***order*** *p if there is a constant C, which may depend on f but not on h, such that*

$$\left| \int_a^b f(x)\,\mathrm{d}x - Q_h(f) \right| \leq Ch^p,$$

for sufficiently smooth functions f.

Theorem 3.3. *Considering the setting of Definition 3.6, suppose that the composite quadrature rule $Q_h(f)$ has degree of exactness r on a single sub-interval. Then $Q_h(f)$ is of order $r+1$. More precisely, for every function f class C^{r+1}, it holds that*

$$\left| \int_a^b f(x)\,\mathrm{d}x - Q_h(f) \right| \leq C \max_{x\in[a,b]} |f^{(r+1)}(x)| h^{r+1}, \tag{3.13}$$

with $C = \frac{(b-a)}{2^r(r+1)!}$.

Proof. We $Q_h(f) = \sum_{i=1}^n Q^{(i)}(f)$, where $Q^{(i)}$ is the quadrature formula applied to the i^{th} sub-interval. Let us consider a sub-interval $I_i = [x_{i-1}, x_i]$ and the Taylor series of the function f around the midpoint $c_i = \frac{x_{i-1}+x_i}{2}$ up to order $r+1$:

$$f(x) = \underbrace{f(c_i) + f'(c_i)(x-c_i) + \ldots + \frac{f^{(r)}(c_i)}{r!}(x-c_i)^r}_{T_f^r(x)} + \underbrace{\frac{f^{(r+1)}(\xi_i)}{(r+1)!}(x-c_i)^{r+1}}_{R_f^r(x)}$$

for some $\xi_i \in (c_i, x)$. Here, T_f^r denotes the degree r Taylor polynomial and R_f^r denotes the residual.

Because of the assumed exactness property, we have that $Q^{(i)}(T_f^r) = \int_{x_{i-1}}^{x_i} T_f^r(x)\,\mathrm{d}x$. Therefore,

$$\begin{aligned}
\left| \int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x - Q^{(i)}(f) \right| &= \left| \int_{x_{i-1}}^{x_i} R_f^r(x)\,\mathrm{d}x - Q^{(i)}(R_f^r) \right| \\
&\leq \left| \int_{x_{i-1}}^{x_i} R_f^r(x)\,\mathrm{d}x \right| + \left| Q^{(i)}(R_f^r) \right| \\
&\leq \frac{(h/2)^{r+1}}{(r+1)!} \max_{x\in[x_{i-1},x_i]} |f^{(r+1)}(x)| \left(\left| \int_{x_{i-1}}^{x_i} 1\,\mathrm{d}x \right| + |Q^{(i)}(1)| \right) \\
&= \frac{h^{r+1}}{2^{r+1}(r+1)!} \max_{x\in[x_{i-1},x_i]} |f^{(r+1)}(x)| 2h.
\end{aligned}$$

Finally,

$$\begin{aligned}\left|\int_a^b f(x)\,\mathrm{d}x - Q_h(f)\right| &= \left|\sum_{i=1}^{n}\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x - Q^{(i)}(f)\right| \\ &\leq \sum_{i=1}^{n}\left|\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x - Q^{(i)}(f)\right| \leq \frac{h^{r+1}}{2^r(r+1)!}\max_{x\in[a,b]}|f^{(r+1)}(x)|\sum_{i=1}^{n} h \\ &= \frac{(b-a)}{2^r(r+1)!}\max_{x\in[a,b]}|f^{(r+1)}(x)|h^{r+1}\end{aligned}$$

which proves the result. □

Using Theorem 3.3, we can establish the following table:

	comp. midpoint	comp. trapezoidal	comp. Simpson
order	2	2	4

Example 3.4. *Let us compute the integral*

$$I = \int_0^{\frac{\pi}{2}} \frac{\sin(x)\cos^3(x)}{4-\cos^2(2x)}\,\mathrm{d}x. \tag{3.14}$$

After some very long calculations, one finds that $I = \frac{\log(3)}{16}$. *The following* **Python** *code uses the composite trapezoidal formula, with an increasing number of sub-intervals* $n = 2, 4, 8, \ldots, 1024$, *to approximate* I.

```
import numpy as np
from ch3_trap import trap

f = lambda x: (np.sin(x) * np.cos(x) ** 3) / (4 - np.cos(2 * x) ** 2)
a = 0
b = np.pi / 2
Iex = np.log(3) / 16
print("Iex=", Iex)
Qhtrap = np.array([])
errQhtrap = np.array([])
N = np.array([2**i for i in range(1, 11)])
h = (b - a) / N
for n in N:
    Q = trap(a, b, n, f)
    print("n=%4d" % n, "  Qh=", Q)
    Qhtrap = np.append(Qhtrap, Q)

# OUTPUT
# Iex= 0.06866326804175686
# n=   2   Qh= 0.04908738521234053
# n=   4   Qh= 0.06421229026829854
# n=   8   Qh= 0.06758370289650516
# n=  16   Qh= 0.06839501624127321
```

```
# n=  32   Qh= 0.06859630316226781
# n=  64   Qh= 0.06864653288981529
# n= 128   Qh= 0.0686590846320792
# n= 256   Qh= 0.06866222221296711
# n= 512   Qh= 0.06866300658603605
# n=1024   Qh= 0.06866320267791895
```

As expected, the approximation becomes increasingly accurate as the number of sub-intervals increases. When using $n = 1024$, we obtain 6 exact significant decimal digits. As the exact value of the integral is known, we can also compute the error committed by the composite trapezoid formula:

```
errQhtrap = abs(Iex - Qhtrap)
for i in range(0, len(errQhtrap)):
    print("n=%4d" % N[i], "  err=%2.16f" % errQhtrap[i])

# OUTPUT
# n=   2   err=0.0195758828294163
# n=   4   err=0.0044509777734583
# n=   8   err=0.0010795651452517
# n=  16   err=0.0002682518004836
# n=  32   err=0.0000669648794891
# n=  64   err=0.0000167351519416
# n= 128   err=0.0000041834096777
# n= 256   err=0.0000010458287898
# n= 512   err=0.0000002614557208
# n=1024   err=0.0000000653638379
```

When doubling n (which means that h is halved), the error is approximately divided by 4, confirming that the composite trapezoid formula is of second order. This becomes even clearer in a loglog plot of the error:

```
import matplotlib.pyplot as plt

plt.loglog(h, errQhtrap, "b-", linewidth=2)
plt.loglog(h, h**2, "k--", linewidth=2)
plt.loglog(h, h**4, "k-.", linewidth=2)
plt.grid(True)
plt.legend(["err. trap", "order 2", "order 4"])
```

Repeating the same calculations for the composite midpoint and Simpson formulas results in Figure 3.3. The graph clearly reflects that the composite midpoint and trapezoidal formulas are of second order, whereas Simpson is of fourth order. Moreover, we notice that the error from the trapezoidal formula is higher (approximately by a factor 2) than the error from the midpoint formula.

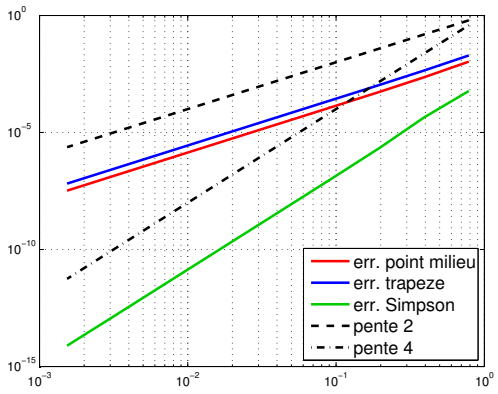


Figure 3.3: Error of the composite midpoint (red), trapezoidal (blue) and Simpson (green) formulas vs. h when approximating the integral (3.14).

3.2.2 Richardson extrapolation

The Richardson extrapolation method is a general technique for increasing the accuracy of approximation formulas. We will apply it to composite quadrature formulas, but it can also be applied to many other situations (such as numerical differentiation or interpolation).

To approximate $I = \int_a^b f(x)\,\mathrm{d}x$, let us consider a composite quadrature formula $Q_h(f)$ based on a partition of the interval $[a, b]$ in n sub-intervals $I_i = [x_{i-1}, x_i]$, $i = 1, \ldots, n$, of length $h = (b-a)/n$. Let us suppose that the formula is of order p, which allows us to write

$$Q_h(f) = I + Ch^p + O(h^q), \qquad q > p, \tag{3.15}$$

where C is a suitable constant which depends, in general, on a high-order derivative of f.

Example 3.5. *Let us consider the composite midpoint formula:* (3.8)

$$Q_h^{mp}(f) = \sum_{i=1}^{n} hf(c_i), \qquad c_i = \frac{x_i + x_{i-1}}{2}.$$

We use a fourth-order Taylor expansion of f around c_i to conclude that

$$\begin{aligned}\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x &= \int_{x_{i-1}}^{x_i} f(c_i)\,\mathrm{d}x + \int_{x_{i-1}}^{x_i} f'(c_i)(x-c_i)\,\mathrm{d}x + \int_{x_{i-1}}^{x_i} \frac{f''(c_i)}{2}(x-c_i)^2\,\mathrm{d}x \\ &\quad + \int_{x_{i-1}}^{x_i} \frac{f'''(c_i)}{6}(x-c_i)^3\,\mathrm{d}x + \int_{x_{i-1}}^{x_i} O((x-c_i)^4)\,\mathrm{d}x \\ &= hf(c_i) + \frac{f''(c_i)}{2}\frac{h^3}{12} + O(h^5),\end{aligned}$$

where we used the fact that $\int_{x_{i-1}}^{x_i}(x-c_i)^r\,\mathrm{d}x$ *vanishes for every odd* r*. For the composite midpoint formula, this implies*

$$\begin{aligned} Q_h^{mp}(f) - I &= \sum_{i=1}^{n}\left(hf(c_i) - \int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x\right) \\ &= \sum_{i=1}^{n}\left(-\frac{f''(c_i)}{2}\frac{h^3}{12} + O(h^5)\right) = -\frac{1}{24}\left(\sum_{i=1}^{n} hf''(c_i)\right)h^2 + O(h^4), \end{aligned}$$

where we used $\sum_{i=1}^{n} O(h^5) = nO(h^5) = \frac{(b-a)}{h}O(h^5) = O(h^4)$*. Observing that*

$$\sum_{i=1}^{n} hf''(c_i) = Q_h^{mp}(f'') = \int_a^b f''(x)\,\mathrm{d}x + O(h^2),$$

we arrive at

$$Q_h^{mp}(f) = I + Ch^2 + O(h^4), \qquad \text{with } C = -\frac{1}{24}\int_a^b f''(x)\,\mathrm{d}x,$$

which corresponds to expression (3.15) *with* $p = 2$ *and* $q = 4$*.*

For a composite quadrature formula $Q_h(f)$, let us now consider the same quadrature formula but with half the number of sub-intervals (each of length $2h$). By (3.15), we have

$$Q_{2h}(f) = I + C(2h)^p + O(h^q).$$

Multiplying this equation with 2^{-p} and subtracting (3.15), it follows that

$$2^{-p}Q_{2h}(f) - Q_h(f) = (2^{-p} - 1)I + O(h^q),$$

and, therefore,

$$\frac{2^pQ_h(f) - Q_{2h}(f)}{2^p - 1} = I + O(h^q). \tag{3.16}$$

We can then define a new quadrature formula on n sub-intervals,

$$\tilde{Q}_h(f) = \frac{2^pQ_h(f) - Q_{2h}(f)}{2^p - 1},$$

and (3.16) shows that this formula is of order $q > p$. It is quite remarkable that two formulas $Q_h(f)$ and $Q_{2h}(f)$, both of order p, can be combined into a formula of order $q > p$!

The described technique is known as *Richardson extrapolation*. It generally works pretty well, as long as one knows the order p of the reference method. We also notice that for the formula $\tilde{Q}_h(f)$ to be of order $q > p$, we need to require more smoothness from the function f. For example, the composite midpoint formula $Q_h^{mp}(f)$ is of order $p = 2$ if the function is of class C^2 whereas its Richardson's extrapolation will only be of order $q = 4$ if the function is of class C^4 (see Example 3.5).

3.2.3 A posteriori error estimation

It is natural to ask how many sub-intervals are needed for a composite quadrature formula to approximate an integral up to a given tolerance. It is difficult to answer this question exactly as one usually does not know the exact value of the integral to be computed and, hence, we cannot even compute the error exactly, let alone estimating the number of required sub-intervals. However, in an heuristic approach, we can proceed as follows: Compute the quadrature formula $Q_h(f)$ on n sub-intervals, compute the quadrature formula $Q_{2h}(f)$ on $n/2$ sub-intervals, and then use the Richardson extrapolation $\tilde{Q}_h(f)$, which is usually more accurate, as a proxy for the exact value I. This idea leads to the *error indicator*

$$\eta_h = |\tilde{Q}_h(f) - Q_h(f)|.$$

In fact, we have

$$|I - Q_h(f)| \leq |I - \tilde{Q}_h(f)| + |\tilde{Q}_h(f) - Q_h(f)| = \underbrace{\eta_h}_{O(h^p)} + O(h^q) \approx \eta_h.$$

If the error indicator does not satisfy the given tolerance, we then double the number of sub-intervals and continue until the estimated error is smaller than the fixed tolerance. This procedure is summarized in the following algorithm.

Algorithm 3.1: Adaptive quadrature based on Richardson extrapolation

Data: $f(x)$, $[a, b]$, n_0, tol
Result: I, err, n
$h = (b - a)/n_0$; `// $n_0$: initial number of sub-intervals`
$I_0 = Q_h(f)$; `// initial estimation of` I
$k = 0$; err $=$ tol $+ 1$;
while err $>$ tol **do**
  $k = k + 1$; $h = h/2$;
  $I_k = Q_h(f)$; `// new approximation of` I
  $\tilde{I}_k = \frac{2^p I_k - I_{k-1}}{2^p - 1}$; `// Richardson extrapolation`
  err $= |\tilde{I}_k - I_k|$;
end
$I = \tilde{I}_k$, $n = (b - a)/h$.

Chapter 4

Linear systems – direct methods

In this chapter, we are interested in the numerical solution of a linear system of n equations in n unknowns $x_1, \ldots, x_n$, which takes the form:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n. \end{cases} \tag{4.1}$$

If we define the vectors $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ and the matrix $A \in \mathbb{R}^{n \times n}$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \qquad A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix},$$

then the system (4.1) can be written on the ore compact form

$$A\mathbf{x} = \mathbf{b}.$$

4.1 Triangular systems

Some linear systems which are particularly simple to solve. Let us consider, for example, a **lower triangular linear system**

$$\begin{cases} a_{11}x_1 & & & & = b_1 \\ a_{21}x_1 & + a_{22}x_2 & & & = b_2 \\ a_{31}x_1 & + a_{32}x_2 & + a_{33}x_3 & & = b_3 \\ \ldots & & & & \\ a_{n1}x_1 & + a_{n2}x_2 & + \ldots & + a_{nn}x_n & = b_n \end{cases} \tag{4.2}$$

with $a_{ii} \neq 0$ for $i = 1, \ldots, n$. To solve this system we can start from the first equation

$$x_1 = \frac{b_1}{a_{11}},$$

then move to the second equation

$$x_2 = \frac{1}{a_{22}} \left(b_2 - a_{21} x_1 \right),$$

and so on, until arriving at the last equation to determine x_n. This idea results in the following algorithm, called *forward substitution*:

Algorithm 4.1: Forward substitution algorithm

for $i = 1, \ldots, n$ **do**
  $x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right)$;
end

Likewise, if we have an **upper triangular system**

$$\begin{cases} a_{11}x_1 + & a_{12}x_2 + & \ldots + & a_{1,n-1}x_{n-1} + & a_{1n}x_n = & b_1 \\ & a_{22}x_2 + & \ldots + & a_{2,n-1}x_{n-1} + & a_{2n}x_n = & b_2 \\ & & & & \ldots & \\ & & & a_{n-1,n-1}x_{n-1} + & a_{n-1,n}x_n = & b_{n-1} \\ & & & & a_{nn}x_n = & b_n \end{cases} \tag{4.3}$$

with $a_{ii} \neq 0$ for $i = 1, \ldots, n$, we can start by solving the last equation $x_n = b_n / a_{nn}$ and work upwards until we reach the first equation. This results in an algorithm called *backward substitution*:

Algorithm 4.2: Backward substitution algorithm

for $i = n, n-1, \ldots, 1$ **do**
  $x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right)$;
end

4.2 Gaussian elimination and LU decomposition

Given any linear system of equations, Gaussian elimination allows us to transform it into an upper triangular system using elementary operations that effect linear combinations of the rows of the matrix. As we saw in the previous section, once we obtain an upper triangular system, it can be easily solved using backward substitution. We will now explain Gaussian elimination algorithm with an example.

Example 4.1. *Let us consider the linear system* $A\mathbf{x} = \mathbf{b}$ *with*

$$A = \begin{bmatrix} 2 & 1 & 0 \\ -4 & 3 & -1 \\ 4 & -3 & 4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ -2 \end{bmatrix}.$$

We let $r_1^{(1)}$, $r_2^{(1)}$, $r_3^{(1)}$ *denote the initial three equations of the system:*

$$\begin{aligned} r_1^{(1)}: \quad & 2x_1 + x_2 = 4 \\ r_2^{(1)}: \quad & -4x_1 + 3x_2 - x_3 = 2 \\ r_3^{(1)}: \quad & 4x_1 - 3x_2 + 4x_3 = -2 \end{aligned}$$

Gaussian elimination transforms these equations, in a systematic way, to an upper triangular system.
1st step*:*

$$\begin{aligned} r_1^{(2)} &\leftarrow r_1^{(1)} && \Longrightarrow 2x_1 + x_2 = 4 \\ r_2^{(2)} &\leftarrow r_2^{(1)} - \underbrace{\left(\frac{\mathbf{-4}}{\mathbf{2}}\right)}_{l_{21}} r_1^{(1)} && \Longrightarrow 5x_2 - x_3 = 10 \\ r_3^{(2)} &\leftarrow r_3^{(1)} - \underbrace{\left(\frac{\mathbf{4}}{\mathbf{2}}\right)}_{l_{31}} r_1^{(1)} && \Longrightarrow -5x_2 + 4x_3 = -10. \end{aligned}$$

2nd step*:*

$$\begin{aligned} r_1^{(3)} &\leftarrow r_1^{(2)} && \Longrightarrow 2x_1 + x_2 = 4 \\ r_2^{(3)} &\leftarrow r_2^{(2)} && \Longrightarrow 5x_2 - x_3 = 10 \\ r_3^{(3)} &\leftarrow r_3^{(2)} - \underbrace{\left(\frac{\mathbf{-5}}{\mathbf{5}}\right)}_{l_{32}} r_2^{(2)} && \Longrightarrow 3x_3 = 0. \end{aligned}$$

We have arrived at an upper triangular system. Notice that the matrix of the system has been transformed in the following way:

$$A = A^{(1)} = \begin{bmatrix} 2 & 1 & 0 \\ -4 & 3 & -1 \\ 4 & -3 & 4 \end{bmatrix} \Longrightarrow A^{(2)} = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 5 & -1 \\ 0 & -5 & 4 \end{bmatrix} \Longrightarrow A^{(3)} = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 5 & -1 \\ 0 & 0 & 3 \end{bmatrix} = U.$$

Therefore, in the first step, the elements below the diagonal in the first column have been annihilated, while in the second step, the elements below the diagonal in the second column has been annihilated.

We now store the multipliers l_{ij} used in every step of the algorithm in the lower triangular part of a matrix L with diagonal elements equal to 1:

$$L^{(1)} = \begin{bmatrix} 1 & & \\ -\frac{4}{2} & 1 & \\ \frac{4}{2} & & 1 \end{bmatrix} \Longrightarrow L^{(2)} = \begin{bmatrix} 1 & & \\ -2 & 1 & \\ 2 & -\frac{5}{5} & 1 \end{bmatrix} \Longrightarrow L^{(3)} = \begin{bmatrix} 1 & & \\ -2 & 1 & \\ 2 & -1 & 1 \end{bmatrix} = L$$

and we notice that $LU = A$.

From the previous example one can deduce the general principle of *Gaussian elimination*, which uses a series of linear row combinations to build an upper triangular matrix that we call U. Moreover, by saving every multiplier used by the linear combinations in a lower triangular matrix L with main diagonal elements equal to 1, we obtain that

$$A = LU.$$

Thus, the matrix A is factored (decomposed) as the product of two triangular matrices. This factorization is called **LU decomposition**. For general n, Gaussian elimination is performed by the following algorithm. Note that I_n denotes the $n \times n$ identity matrix.

Algorithm 4.3: Gaussian elimination (and LU decomposition)

Data: $A = \{a_{ij}\} \in \mathbb{R}^{n\times n}$, $\mathbf{b} = \{b_i\} \in \mathbb{R}^n$
Result: $L, U \in \mathbb{R}^{n\times n}$, $\mathbf{b}^{(n)} \in \mathbb{R}^n$
$A^{(1)} = A$; $L = I_n$;
for $k = 1, \ldots, n-1$ **do** `// algorithm steps`
  for $i = k+1, \ldots, n$ **do** `// loop over the rows`
    $l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$;
    for $j = k+1, \ldots, n$ **do** `// loop over the columns`
      $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$;
    end
    $b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)}$;
  end
end
$U = A^{(n)}$;

Algorithm 4.3 returns matrices L and U such that $A = LU$. Additionally a vector $\mathbf{b}^{(n)}$ is returned that contains the original vector $\mathbf{b}$ modified by the same operations the rows of A are subjected to. Comparing with forward substitution one notes that $\mathbf{b}^{(n)} = L^{-1}\mathbf{b}$. One still needs to solve the upper triangular system $U\mathbf{x} = \mathbf{b}^{(n)}$, which is done by backward substitution.

The computation of $\mathbf{b}^{(n)}$ *during* Algorithm 4.3 is optional. If we only compute the LU decomposition of A, we can still solve the linear system

$$A\mathbf{x} = \mathbf{b} \qquad \Leftrightarrow \qquad LU\mathbf{x} = \mathbf{b}$$

afterwards. Introducing the auxiliary vector $\mathbf{y}$, the linear system can be solved in two steps:

$$\begin{cases} \text{Solve } L\mathbf{y} = \mathbf{b} & \text{(lower triangular system } \rightsquigarrow \text{ forward substitution)} \\ \text{Solve } U\mathbf{x} = \mathbf{y} & \text{(upper triangular system } \rightsquigarrow \text{ backward substitution).} \end{cases}$$

Note that the vector $\mathbf{y}$ corresponds to the vector $\mathbf{b}^{(n)}$ returned by Algorithm 4.3.

4.3 Gaussian elimination with pivoting

Gaussian elimination, as prescribed by Algorithm 4.3, does not always succeed even when A is invertible. The coefficient $a_{kk}^{(k)}$ at the k^{th} step may become zero, which makes it impossible to continue, as we cannot compute the multiplier l_{ik}. The elements $a_{kk}^{(k)}$ are called **pivots**.

In order to avoid zero pivots and continue the algorithm, one can swap rows of the matrix, a technique known as *pivoting*. Let us explain this idea for an example.

Example 4.2. *Let us perform Gaussian elimination for the matrix*

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix}.$$

After the first step of the algorithm, we obtain the following matrices

$$A^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & \boxed{0} & -1 \\ 0 & -6 & -12 \end{bmatrix}, \qquad L^{(1)} = \begin{bmatrix} 1 & & \\ 2 & 1 & \\ 7 & & 1 \end{bmatrix}.$$

As the pivot $a_{22}^{(2)}$ is zero, Gaussian elimination cannot continue.

However, this situation is resolved when swapping the first and second rows in the matrix $A^{(2)}$. Correspondingly, we also need to swap the multipliers contained in in the matrix $L^{(2)}$. This gives

$$\tilde{A}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{bmatrix}, \qquad \tilde{L}^{(1)} = \begin{bmatrix} 1 & & \\ 7 & 1 & \\ 2 & & 1 \end{bmatrix}.$$

The new pivot $\tilde{a}_{22}^{(2)}$ is not zero and the algorithm can continue. By coincidence, the matrix $\tilde{A}^{(2)}$ is already upper triangular and nothing remains to be done. We have reached the final step of the factorization:

$$U = A^{(3)} = \tilde{A}^{(2)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{bmatrix}, \qquad L = L^{(3)} = \begin{bmatrix} 1 & & \\ 7 & 1 & \\ 2 & 0 & 1 \end{bmatrix}.$$

If we now compute the matrix multiplication

$$LU = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 7 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{bmatrix}}_{U} = \begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \\ 2 & 4 & 5 \end{bmatrix},$$

we recover the initial matrix A, but with the second and third rows swapped. This can be addressed by applying the same row permutations to A that we performed during the algorithm.

Let us introduce the ***permutation matrix***

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

This matrix is obtained using the identity matrix and performing a permutation of the second and third rows. Then one easily verifies that

$$PA = LU.$$

The previous example indicates that performing a suitable permutation every time a pivot is zero, then Gaussian elimination terminates successfully for an invertible matrix and we obtain two triangular matrices L and U. Their multiplication is equal to PA, where the permutation matrix P reflects every permutation performed during the algorithm.

In general, if a pivot $a_{kk}^{(k)}$ is zero, we can decide to swap row k with any row $i > k$ that satisfies $a_{ik}^{(k)} \neq 0$. However, from a numerical perspective, a tiny value of $|a_{ik}^{(k)}|$ will blow up the size of the entries in the transformed matrix and lead to massive problems with roundoff error. To make the algorithm numerically robust, one chooses the row r that gives the pivot of maximal magnitude: $|a_{rk}^{(k)}| \geq |a_{ik}^{(k)}|$ for every $i = k, \ldots, n$. In turn, one *always* performs a swap of rows k and r, even when the pivot $a_{kk}^{(k)}$ is not zero. A rigorous formulation of the resulting algorithm is given in Algorithm 4.4. The success of the algorithm is guaranteed (in exact arithmetic).

Theorem 4.1. *Algorithm 4.4 returns for every invertible matrix $A \in \mathbb{R}^{n \times n}$ a lower triangular matrix L (with diagonal entries 1), an upper triangular matrix U, and a permutation matrix P, such that*

$$PA = LU. \tag{4.4}$$

Given the decomposition (4.4), we we now want to solve the linear system $A\mathbf{x} = \mathbf{b}$. We notice that

$$A\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad PA\mathbf{x} = P\mathbf{b} \quad \Leftrightarrow \quad LU\mathbf{x} = P\mathbf{b}.$$

Algorithm 4.4: Gaussian elimination with pivoting

Data: $A = \{a_{ij}\} \in \mathbb{R}^{n\times n}$, $\mathbf{b} = \{b_i\} \in \mathbb{R}^n$
Result: $L, U, P \in \mathbb{R}^{n\times n}$, $\mathbf{b}^{(n)} \in \mathbb{R}^n$
$A^{(1)} = A$; $L = I_n$; $P = I_n$;
for $k = 1, \ldots, n-1$ **do**
  determine r such that $|a_{rk}^{(k)}| = \max_{i=k,\ldots,n} |a_{ik}^{(k)}|$;
  swap rows k and r in the matrices $A^{(k)}$ and P, as well as in the vector $\mathbf{b}^{(k)}$ and the first $k-1$ columns of L;
  for $i = k+1, \ldots, n$ **do**
    $l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$;
    for $j = k+1, \ldots, n$ **do**
      $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$;
    end
    $b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)}$;
  end
end
$U = A^{(n)}$;

We can then solve the linear system using the following operations:

$$\begin{cases} \text{Solve } L\mathbf{y} = P\mathbf{b} & \text{(lower triangular system} \rightsquigarrow \text{ forward substitution)} \\ \text{Solve } U\mathbf{x} = \mathbf{y} & \text{(upper triangular system} \rightsquigarrow \text{ backward substitution).} \end{cases}$$

Again, the vector $\mathbf{y}$ corresponds to the vector $\mathbf{b}^{(n)}$ returned by Algorithm 4.4.

Gaussian elimination algorithm with pivoting is utilized by the `Python` commands numpy.linalg.solve and scipy.linalg.solve, Specifically, one calls x=scipy.linalg.solve(A,b) to solve the linear system $A\mathbf{x} = \mathbf{b}$. One can also explicitly calculate the LU decomposition with pivoting using the command P,L,U = scipy.linalg.lu (A). The following code performs this for the matrix from Example 4.2.

```
import numpy as np
import scipy.linalg as sc

A = np.array([[1, 2, 3], [2, 4, 5], [7, 8, 9]])
P, L, U = sc.lu(A)
print("L: ", L)
print("U: ", U)
print("P: ", P)

# OUTPUT
# L:  [[1.         0.         0.        ]
#      [0.28571429 1.         0.        ]
```

```
#      [0.14285714 0.5        1.        ]]
# U:  [[7.         8.         9.        ]
#      [0.         1.71428571 2.42857143]
#      [0.         0.         0.5       ]]
# P:  [[0. 0. 1.]
#      [0. 1. 0.]
#      [1. 0. 0.]]
```

Note that `Python` used a permutation different from the one we used in Example 4.2. The product LU gives the original matrix A with the first and third rows swapped.

```
print("L*U: ", np.dot(L, U))

# OUTPUT
# L*U:  [[7. 8. 9.]
#        [2. 4. 5.]
#        [1. 2. 3.]]
```

4.4 Memory usage and fill-in

Many engineering applications lead to linear systems of enormous sizes. However, often many of the matrix entries are actually zero. This has important consequences on how such a matrix should be stored in memory as well as on the computational cost of solving a linear system.

Definition 4.1 (**Dense matrix**). *We say that a matrix $A \in \mathbb{R}^{n\times n}$ is* dense *if the number of non-zero entries is of the order of n^2 (which means that a significant fraction of the entries are non-zero).*

Storing a dense matrix requires $O(n^2)$ memory. When using double precision floating point numbers, every entry consumes 64 bits of memory, which corresponds to 8 bytes (as every byte consists of 8 bits). Laptops have a (RAM) memory of a few Giga-bytes. To give a rough idea, let us consider a laptop with 4 GB $\approx 4 \cdot 10^9$ bytes of RAM, which allows us to store at most $5 \cdot 10^8$ double precision floating point numbers. Thus, the absolute maximum on the size of a fully populated matrix that can be stored in memory is $n = \sqrt{5 \cdot 10^8} \approx 20000$. Engineering applications, such as structural analysis of materials or fluid flow simulations, easily lead to linear systems with 10^5 to 10^7 unknowns. These linear systems can only be solved by exploiting the sparsity of the matrices.

Definition 4.2 (**Sparse matrix**). *We say that $A \in \mathbb{R}^{n\times n}$ is a* sparse matrix *if the number of non-zero entries is of the order of n.*

In other words, a sparse matrix has – on average – a constant number of entries per row. It suffices to store the non-zero entries along with their

positions in the matrix. This reduces memory usage to $O(n)$ and it is easily possible to store a $10^6 \times 10^6$ matrix on a laptop if the matrix is sparse.

In `Python`, several formats for storing sparse matrices are available using scipy, depending on in which order the entries and how their positions are stored. Popular choices include:

- Compressed Sparse Row format (`scipy.sparse.csr_matrix`),
- Compressed Sparse Column format (`scipy.sparse.csr_matrix`),
- COOrdinate format (`scipy.sparse.coo_matrix`).

For example, the following code stores the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 5 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}$$

in the Compressed Sparse Row format.

```
import scipy.sparse as sp
import numpy as np

A = np.array([[1, 0, 0, 5],
              [-1, 0, 1, 0],
              [0, 0, 3, 0],
              [0, 1, 0, -1]])
A = sp.csr_matrix(A)
print(A)

# OUTPUT
# (0, 0)  1
# (0, 3)  5
# (1, 0)  -1
# (1, 2)  1
# (2, 2)  3
# (3, 1)  1
# (3, 3)  -1
```

It can be seen that `Python` only stores the non-zero entries of the matrix along with their positions.

Definition 4.3 (**Banded matrix**). *$A \in \mathbb{R}^{n\times n}$ is called a* banded matrix with bandwidth *K if its entries a_{ij} satisfy*

$$a_{ij} = 0, \quad \text{if} \quad |j - i| > K.$$

Every row of a banded matrix with bandwidth K contains at most $2K+1$ non-zero entries. For $K = 1$, such a matrix is called *tridiagonal*. A band matrix is a particularly simple (but still important) case of a sparse matrix, where the non-zero elements are concentrated around the diagonal. Storage as a sparse matrix requires $O(Kn)$ (instead of $O(n^2)$) memory.

The `Python` command matplotlib.pyplot.spy(A) visualizes the non-zero entries of a matrix; see Figure 4.1 for examples.

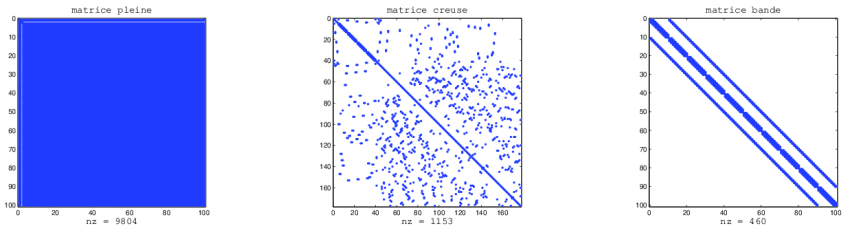


Figure 4.1: Example of a full (left), sparse (middle) and banded (right) matrices. Visualization obtained using the command spy.

The memory needed to store the factors L, U of an LU decomposition for a sparse matrix is difficult to predict. Obviously, the factors for a full matrix can be expected to be full (below and above the diagonal, respectively). On the other hand, the factors for a sparse matrix are not necessarily sparse, due to a phenomenon called *fill-in*. For example, Figure 4.2 (middle) shows that the factors lose a lot of the sparsity of the original matrix from Figure 4.1 (middle). For a banded matrix A, it can be shown that the factors L and U are zero below and above the bands of A, respectively. At the same time, additional zero structure within the bands is not necessarily preserved by the LU decomposition; see Figure 4.2 (right). In many practically relevant

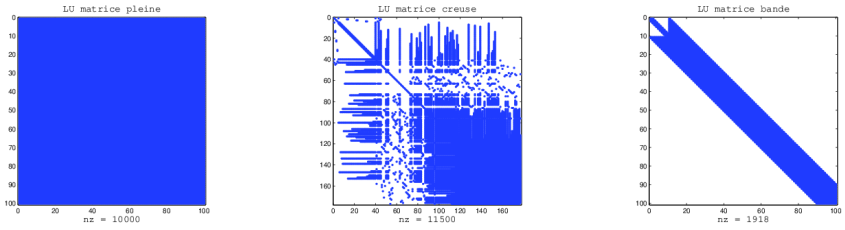


Figure 4.2: Non-zero entries of the LU decompositions of the matrices from Figure 4.1. L corresponds to the lower triangular part and U to the upper triangular part of the visualized matrices. Left: full matrix; middle: sparse matrix; right: banded matrix.

situations, fill-in can be reduced (sometimes dramatically) , to a certain extent by reordering the rows and columns of the matrix A *prior* to performing an LU decomposition. The `Python` command scipy.sparse.linalg.spsolve effects such a reordering, using UMFPACK by default. Other popular soft-

ware packages for effecting such sparse direct LU decompositions include MUMPS, Pardiso, and SuperLU.

4.5 Computational cost of LU decompositions

Processors on laptops nowadays work with a frequency of a few Giga-Hertz, allowing one to perform a $O(10^9)$ operations per second. In the following, we will perform rough estimates of the computational cost for some typical linear algebra operations.

- **Scalar product**: Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, the scalar product

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{y}^\top \mathbf{x} = \sum_{i=1}^{n} x_i y_i$$

  requires n multiplications and $n-1$ additions. Hence, the number of elementary operations is of the order of n (more precisely, $2n-1$) and we say that the computational cost is $O(n)$. For $n = 1000$, this translates into approximately 10^{-6} seconds computational time, as the computer can perform $O(10^9)$ operations per second.

- **Product of a dense matrix with a vector**: Given $\mathbf{x} \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$, we want to compute

$$\mathbf{y} = A\mathbf{x}, \qquad \Longrightarrow \qquad y_i = \sum_{j=1}^{n} A_{ij} x_j, \;\; i = 1, \ldots, n.$$

  Computing each entry y_i requires n multiplications and $(n-1)$ additions. As there are n entries in $\mathbf{y}$, the computational cost of this operation is $O(n^2)$. For $n = 1000$, this translates into approximately 10^{-3} seconds computational time.

We now consider the LU decomposition of a dense matrix, using Algorithm 4.3 (or Algorithm 4.4). The cost is dominated by the operation

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)},$$

which contains a multiplication and a subtraction, that is, two elementary operations. This instruction is inside 3 nested loops (k, i, and j); we therefore arrive at

$$\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} \sum_{j=k+1}^{n} 2 = O(n^3)$$

operations. For $n = 1000$, this translates into a computational time in the order of 1 second. Due to the cubic dependence, this number increases

quickly with n. For example, for $n = 10^4$, the computational time becomes $\sim 10^3$ seconds, that is, in the order of 15 minutes. For a banded matrix with bandwidth K, we can limit the two internal loops in Algorithm 4.3 to K entries and, in turn, the computational cost reduces to $O(nK^2)$.

The following table summarizes the computational cost and memory requirements of an LU decomposition for an $n \times n$ matrix:

	compt cost	memory
dense matrix	$O(n^3)$	$O(n^2)$
banded matrix (bandwidth K)	$O(nK^2)$	$O(nK)$

4.6 Effects of round-off errors

Gaussian elimination (with pivoting) yields the exact solution of a linear system in a finite number of operations. A computer, however, executes all operations inexactly and uses floating point representation for real numbers, which introduces small round-off errors of the order of 10^{-16} (in double precision).

In this section, we aim at giving some insights on the effect of these round-off errors on the solution obtained by the computer. We formalize this question as follows. Assume we wish to solve

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

which will be called the "exact system". Already the storage of the vector $\mathbf{b}$ and the matrix A is usually not exact because their entries need to be rounded in order to fit the floating point format. Hence, the computer solves a slightly modified system

$$\hat{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}, \tag{4.5}$$

which will be called the "perturbed system". Apart from rounding the input, Gaussian elimination introduces additional roundoff error in every operations. Backward error analysis allows one to push this error back to the original system, allowing one to view the computed solution as the exact solution of the perturbed (4.5). For a good (backward stable) algorithm, like Gaussian elimination with pivoting, we expect that this perturbed system remains very close to the exact system. More specifically, we may assume that

$$\hat{b}_i = b_i(1 + \epsilon_i), \qquad \text{where } \epsilon_i \text{ is of order of } 10^{-16}.$$

Likewise, the entries of $\hat{A}$ satisfy

$$\hat{a}_{ij} = a_{ij}(1 + \eta_{ij}), \qquad \text{with } \eta_{ij} \text{ of order of } 10^{-16}.$$

One may conclude that the solution $\hat{\mathbf{x}}$ of the slightly perturbed system (4.5) is close to the exact solution $\mathbf{x}$. This is indeed often but not

always the case, depending on the matrix A. The following example shows that things may go wrong.

Example 4.3. *Let us consider the exact linear system*

$$A\mathbf{x} = \mathbf{b} \qquad \Longleftrightarrow \qquad \begin{bmatrix} 1 & 10^{-16} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

with the solution $(x_1, x_2) = (1, 0)$*, and the perturbed system*

$$\hat{A}\hat{\mathbf{x}} = \hat{\mathbf{b}} \qquad \Longleftrightarrow \qquad \begin{bmatrix} 1 & 10^{-16} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \end{bmatrix} = \begin{bmatrix} 1 + 10^{-16} \\ 1 \end{bmatrix}.$$

Note that we only perturbed the first entry of $\mathbf{b}$*. The solution of the perturbed system is* $(\hat{x}_1, \hat{x}_2) = (1, 1)$*, that is, the second entry of the solution is completely wrong! In other words, the small perturbation* 10^{-16} *of the right-hand side term has been amplified tremendously!*

To better understand the phenomenon observed in Example 4.3 and analyze the error $\mathbf{x} - \hat{\mathbf{x}}$, we first introduce some notation. Let us recall that the Euclidean norm of a vector $\mathbf{x}$ is $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$. We aim at estimating the *relative error* $\|\hat{\mathbf{x}} - \mathbf{x}\|/\|\mathbf{x}\|$. For this purpose, we need to generalize the notion of norm to matrices.

Definition 4.4 (**Norm of a matrix**)**.** *Given a matrix* $A \in \mathbb{R}^{m\times n}$ *(not necessarily square), we define the* (spectral) norm of A *as*

$$\|A\| = \sup_{\substack{\mathbf{x}\in\mathbb{R}^n \\ \mathbf{x}\neq\mathbf{0}}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}.$$

This definition of norm implies, in particular, that

$$\|A\mathbf{x}\| \leq \|A\|\|\mathbf{x}\|, \qquad \forall \mathbf{x} \in \mathbb{R}^n.$$

Given square symmetric matrix $B \in \mathbb{R}^{n\times n}$, let $\lambda_i(B) \in \mathbb{R}$, $i = 1, \ldots, n$, denote the eigenvalues of B. We set $\lambda_{\max}(B) = \max_{i=1,\ldots n} \lambda_i(B)$ and $\lambda_{\min}(B) = \min_{i=1,\ldots n} \lambda_i(B)$, the maximum and minimum eigenvalues, respectively. We have the following characterization of the norm of a matrix.

Lemma 4.2. *For any matrix* $A \in \mathbb{R}^{m\times n}$*, it holds that* $\|A\| = \sqrt{\lambda_{\max}(A^\top A)}$*. If* A *is square and invertible, then*

$$\|A^{-1}\| = \sqrt{\lambda_{\max}(A^{-\top}A^{-1})} = \frac{1}{\sqrt{\lambda_{\min}(A^\top A)}}.$$

Definition 4.5 (**Condition number**)**.** *The (spectral)* condition number *of a square and invertible matrix* A *is defined as*

$$\kappa(A) = \|A^{-1}\|\|A\| = \frac{\sqrt{\lambda_{\max}(A^\top A)}}{\sqrt{\lambda_{\min}(A^\top A)}}.$$

In `Python`, the norm and condition number of a matrix are computed using the commands numpy.linalg.norm and numpy.linalg.cond, respectively.

If A is a symmetric matrix, that is, $A^\top = A$, then

$$\lambda_i(A^\top A) = \lambda_i(A^2) = \lambda_i(A)^2, \qquad i = 1, \ldots, n.$$

A symmetric matrix A is positive definite if and only if all its eigenvalues are positive. In this case, it follows that

$$\kappa(A) = \lambda_{\max}(A)/\lambda_{\min}(A).$$

We now aim at analyzing the relative error $\|\hat{\mathbf{x}} - \mathbf{x}\|/\|\mathbf{x}\|$. For simplicity, we will analyze the case when only perturbations in the right-hand side $\mathbf{b}$ are allowed (that is, $\eta_{ij} = 0$). The findings are nearly identical when perturbations in A are allowed as well. We have

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ A\hat{\mathbf{x}} &= \hat{\mathbf{b}} \end{aligned} \qquad \Longrightarrow \qquad A(\hat{\mathbf{x}} - \mathbf{x}) = \hat{\mathbf{b}} - \mathbf{b}\,,$$

and therefore

$$\hat{\mathbf{x}} - \mathbf{x} = A^{-1}(\hat{\mathbf{b}} - \mathbf{b}) \quad \Longrightarrow \quad \|\hat{\mathbf{x}} - \mathbf{x}\| \le \|A^{-1}\|\|\hat{\mathbf{b}} - \mathbf{b}\|,$$

where

$$\|\hat{\mathbf{b}} - \mathbf{b}\| = \big(\sum_{i=1}^{n} b_i^2 \epsilon_i^2\big)^{1/2} \le \max_{i=1,\ldots,n} |\epsilon_i| \|\mathbf{b}\|.$$

On the other side,

$$\|\mathbf{b}\| \le \|A\|\|\mathbf{x}\| \quad \Longrightarrow \quad \frac{1}{\|\mathbf{x}\|} \le \frac{\|A\|}{\|\mathbf{b}\|}.$$

If we multiply the two inequalities, we get

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \le \|A\|\|A^{-1}\| \frac{\|\hat{\mathbf{b}} - \mathbf{b}\|}{\|\mathbf{b}\|} \le \kappa(A) \max_{i=1,\ldots,n} |\epsilon_i|. \tag{4.6}$$

In summary, we have proved the following result.

Lemma 4.3. *For an invertible matrix $A \in \mathbb{R}^{n\times n}$, let $\mathbf{x} \in \mathbb{R}^n$ and $\hat{\mathbf{x}} \in \mathbb{R}^n$ denote the solutions of the linear systems $A\mathbf{x} = \mathbf{b}$ and $A\hat{\mathbf{x}} = \hat{\mathbf{b}}$, respectively, where $\hat{b}_i = b_i(1 + \epsilon_i)$ for $i = 1, \ldots, n$. Then*

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \le \kappa(A)\, \epsilon_{\max}, \tag{4.7}$$

where $\epsilon_{\max} = \max_{i=1,\ldots,n} |\epsilon_i|$.

The inequality (4.7) shows that the condition number of the matrix A plays the role of the *amplification factor* of how round-off errors impact the accuracy of the solution.

Example 4.4. *Coming back to Example 4.3, let us compute the condition number of A. We have*

$$A^\top A = \begin{pmatrix} 1 & 1 \\ 10^{-16} & 0 \end{pmatrix} \begin{pmatrix} 1 & 10^{-16} \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 10^{-16} \\ 10^{-16} & 10^{-32} \end{pmatrix}$$

and

$$\det(A^\top A - \lambda I) = \lambda^2 - (2 + 10^{-32})\lambda + 10^{-32}.$$

Hence, the two eigenvalues of $A^\top A$ are

$$\lambda_{1,2}(A^\top A) = \frac{1}{2}(2 + 10^{-32} \pm \sqrt{4 + 10^{-64}})$$

and therefore

$$\lambda_{\max}(A^\top A) \approx 2, \qquad \lambda_{\min}(A^\top A) \approx \frac{1}{2}10^{-32}.$$

We conclude that the condition number of A is

$$\kappa(A) = \sqrt{\frac{\lambda_{\max}(A^\top A)}{\lambda_{\min}(A^\top A)}} \approx 2 \cdot 10^{16}$$

which explains the bad results reported in Example 4.3.

Chapter 5

Linear systems – iterative methods

In the previous chapter we studied Gaussian elimination for solving a linear system $A\mathbf{x} = \mathbf{b}$. However, we also saw that the computational cost and the memory requirements of this method can be excessive as the size of A increases. Even for sparse matrices, fill-in may severely challenge the feasibility of LU decompositions.

In this chapter, we will study *iterative methods* for (approximately) solving linear systems, which constitute an alternative to Gaussian elimination for large-scale problems. The general idea is to build a sequence of vectors $\mathbf{x}^{(k)}$ that converges to the solution $\mathbf{x}$ of the system $A\mathbf{x} = \mathbf{b}$:

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}.$$

5.1 Richardson methods

A general procedure to build iterative methods is to choose an *invertible* matrix P (for which linear systems are very easy to solve) and incorporate this matrix into $A\mathbf{x} = \mathbf{b}$ by re-writing the system in the equivalent form

$$P\mathbf{x} = (P - A)\mathbf{x} + \mathbf{b}.$$

Given an arbitrary initial vector $\mathbf{x}^{(0)}$, we perform the iteration

$$P\mathbf{x}^{(k+1)} = (P - A)\mathbf{x}^{(k)} + \mathbf{b}, \qquad k = 0, 1, \ldots, \tag{5.1}$$

which requires to solve a linear system with P in every iteration. *If* the sequence $\{\mathbf{x}^{(k)}\}_{k\geq 0}$ converges to a vector $\mathbf{x}_\infty$, $\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}_\infty$, then it necessarily follows that

$$P\mathbf{x}_\infty = (P - A)\mathbf{x}_\infty + \mathbf{b} \qquad \Longleftrightarrow \qquad A\mathbf{x}_\infty = \mathbf{b}.$$

In other words, $\mathbf{x}_\infty = \mathbf{x}$ is the solution of the original linear system.

The iteration (5.1) can be rearranged as

$$\begin{aligned} &\text{Solve } P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}, \qquad \mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}, \\ &\text{Set } \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{z}^{(k)}, \end{aligned} \tag{5.2}$$

leading to the following algorithm:

Algorithm 5.1: Richardson method (without stopping criterion)

Given $\mathbf{x}^{(0)}$ and $P \in \mathbb{R}^{n\times n}$ invertible;
for $k = 0, 1, \ldots$ **do**
  compute $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$;
  solve the linear system $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$;
  compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{z}^{(k)}$;
end

Methods of the form (5.2) are called *Richardson methods*. The matrix P is called *preconditioner* and the vector $\mathbf{r}^{(k)}$ is called the *residual* of $\mathbf{x}^{(k)}$. Note that $\mathbf{r}^{(k)} = 0$ if and only if $\mathbf{x}^{(k)}$ is the exact solution of the system. Normally, $\mathbf{r}^{(k)} \neq 0$, and the norm of $\mathbf{r}^{(k)}$ can be viewed as a measure of how far the vector $\mathbf{x}^{(k)}$ is from the exact solution. This will be used for stopping Algorithm 5.1 in Section 5.4.

Note that the Richardson method can be viewed as a *fixed-point method* (as the ones studied in Chapter 1 for $n = 1$): After rewriting the equation $\mathbf{f}(\mathbf{x}) = A\mathbf{x} - \mathbf{b} = \mathbf{0}$ in the equivalent form

$$\mathbf{x} = \boldsymbol{\phi}(\mathbf{x}) = P^{-1}[(P - A)\mathbf{x} + \mathbf{b}],$$

the Richardson method amounts to performing the fixed-point iteration $\mathbf{x}^{(k+1)} = \boldsymbol{\phi}(\mathbf{x}^{(k)})$.

5.1.1 Computational cost

Let us discuss the cost of Algorithm 5.1, first for the case of a *dense* matrix A. The most costly operations in every iteration are:

- The matrix-vector product $A\mathbf{x}^{(k)}$, requiring $O(n^2)$ operations.
- The solution of the linear system $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$. Its cost entirely depends on the choice of the preconditioner P. For example, for a diagonal matrix P, the computational cost is $O(n)$, while for a triangular matrix, the cost is $O(n^2)$.

Unless P is unfortunately chosen, we expect that the cost of one iteration is $O(n^2)$. Thus, if we obtain a sufficiently accurate approximation to the solution in much less than n iterations, then the total cost will be smaller than the $O(n^3)$ needed by Gaussian elimination.

When A is sparse then the cost of a matrix-vector product reduces to $O(n)$. In this case, the most common choices for P (see below) also lead to a cost of $O(n)$ when solving a linear system with P. Hence, the cost of one step of the iteration reduces to $O(n)$. *If* the Richardson method does not converge too slowly, it offers a cheap alternative to direct methods. The convergence speed will be analyzed in Section 5.3.

5.2 Jacobi and Gauss-Seidel methods

The Jacobi and Gauss-Seidel methods are particular instances of the Richardson method.

In the **Jacobi method** the preconditioner P is a diagonal matrix containing the diagonal of A (in `Python` P=numpy.diag(numpy.diag(A))):

$$P = \begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix}.$$

The $(k+1)^{\text{th}}$ iteration of the Jacobi method amounts to solving

$$\begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix} \begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{bmatrix} = - \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & & \vdots \\ \vdots & & \ddots & \\ & \cdots & a_{n,n-1} & 0 \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

The i^{th} component of the vector $\mathbf{x}^{(k+1)}$ can be computed using

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \Big(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \Big), \qquad i = 1, \ldots, n\,, \tag{5.3}$$

provided that $a_{ii} \neq 0$ holds for $i = 1, \ldots, n$.

The **Gauss-Seidel method** uses the preconditioner P containing the lower triangular part of A, including its diagonal (in `Python` P=numpy.tril(A)):

$$P = \begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{n1} & \cdots & \cdots & a_{nn} \end{bmatrix}.$$

The $(k+1)^{\text{th}}$ iteration of the Gauss-Seidel method amounts to solving

$$\begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{n1} & \cdots & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{bmatrix} = - \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ & 0 & \ddots & \vdots \\ & & \ddots & a_{n-1,n} \\ & & & 0 \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

This lower triangular linear system can be solved by forward substitution. In turn, the i^{th} component of the vector $\mathbf{x}^{(k+1)}$ is given by

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \qquad i = 1, \ldots, n, \quad (5.4)$$

again assuming that $a_{ii} \neq 0$ for $i = 1, \ldots, n$.

The Gauss-Seidel method is very similar to the Jacobi method, with the major difference that that while computing a new component $x_i^{(k+1)}$ we use the updated components $x_j^{(k+1)}$, for $j < i$, instead of the old components $x_j^{(k)}$.

5.3 Convergence analysis

The Jacobi and Gauss-Seidel methods are *not* guaranteed converge for every invertible matrix A. To gain more insight in this matter, we will now analyze the general class of Richardson methods

$$P\mathbf{x}^{(k+1)} = (P - A)\mathbf{x}^{(k)} + \mathbf{b} \quad (5.5)$$

with an invertible preconditioned P. We let

$$\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$$

denote the error after k iterations. We want to understand under which conditions $\mathbf{e}^{(k)}$ converges to zero as $k \to \infty$, which means that all its entries converge to zero. Equivalently, the norm of $\mathbf{e}^{(k)}$ converges to zero.

We use that the exact solution satisfies the system

$$P\mathbf{x} = (P - A)\mathbf{x} + \mathbf{b}\,, \quad (5.6)$$

which is equivalent to the initial system $A\mathbf{x} = \mathbf{b}$. Subtracting (5.5) from (5.6) gives $P\mathbf{e}^{(k+1)} = (P - A)\mathbf{e}^{(k)}$ or, equivalently,

$$\mathbf{e}^{(k+1)} = P^{-1}(P - A)\mathbf{e}^{(k)} = \underbrace{(I - P^{-1}A)}_{B}\,\mathbf{e}^{(k)}.$$

Definition 5.1. *The matrix* $B = (I - P^{-1}A)$ *is the iteration matrix of* (5.5).

Recalling Definition 4.4 of a *matrix norm*, we have

$$\begin{aligned}\|\mathbf{e}^{(k+1)}\| &\leq \|B\|\|\mathbf{e}^{(k)}\| \leq \|B\|^2\|\mathbf{e}^{(k-1)}\| \\ &\leq \cdots \leq \|B\|^{k+1}\|\mathbf{e}^{(0)}\|.\end{aligned}$$

Hence, the error converges to zero if $\|B\| < 1$. We proved the following result.

Theorem 5.1. *The Richardson method* (5.5) *with the iteration matrix* $B = (I - P^{-1}A)$ *converges for any initial vector* $\mathbf{x}^{(0)}$ *if* $\|B\| < 1$. *Moreover,*

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq \|B\|^k\|\mathbf{x} - \mathbf{x}^{(0)}\|. \tag{5.7}$$

Notice the similarity between the convergence condition $\|B\| < 1$ for the Richardson method and the condition $|\phi'(\alpha)| < 1$ for the local convergence of a fixed-point method (see Chapter 1). However, in contrast fixed-point methods for a nonlinear equation, the method (5.5) converges for any initial vector $\mathbf{x}^{(0)}$ if $\|B\| < 1$.

The quantity $\|B\|$ gives an indication of the convergence speed because, according to (5.7), a small value of $\|B\|$ leads to fast convergence.

Let us emphasize that $\|B\| < 1$ is sufficient but *not* necessary for convergence. A *necessary and sufficient* condition can be formulated in terms of the spectral radius.

Definition 5.2 (**Spectral radius**). *Let* $\lambda_i(B)$, $i = 1, \ldots, n$, *denote the eigenvalues of a square matrix* $B \in \mathbb{R}^{n\times n}$. *Then*

$$\rho(B) = \max_{i=1,\ldots,n} |\lambda_i(B)| \tag{5.8}$$

is called the spectral radius *of* B.

Theorem 5.2. *The Richardson method* (5.5) *converges for every initial vector if and only if the iteration matrix* $B = (I - P^{-1}A)$ *satisfies* $\rho(B) < 1$.

Based on the theory developed, it can be shown that the Jacobi and Gauss-Seidel methods are guaranteed to converge if A is symmetric positive definite or strictly diagonal dominant (that is, $|a_{ii}| > \sum_{j\neq i} |a_{ij}|$).

5.4 Error control and stopping criterion

We now supplement Algorithm 5.1 with a stopping criterion. Ideally, one would like to stop the iterations when the norm of the error $\|\mathbf{e}^{(k)}\| = \|\mathbf{x} - \mathbf{x}^{(k)}\|$ is smaller than a given tolerance. Unfortunately, this requires knowledge of the exact solution. Instead, we can use the residual

$$\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$$

to control convergence. Because of $\|\mathbf{r}^{(k)}\| = \|A\mathbf{x} - A\mathbf{x}^{(k)}\| \leq \|A\| \, \|\mathbf{x} - \mathbf{x}^{(k)}\|$, the norm of $\mathbf{r}^{(k)}$ is small when $\mathbf{x}^{(k)}$ is close to the exact solution. Specifically, we verify that the residual is small relative to the right-hand-side:

$$\textbf{stopping criterion:} \quad \frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|} \leq \mathsf{tol}. \tag{5.9}$$

This leads to the following algorithm:

Algorithm 5.2: Richardson method (with stopping criterion)

Data: A, $\mathbf{b}$, $\mathbf{x}^{(0)}$, P, tol
Result: $\mathbf{x}$, res, niter
$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$; $k = 0$;
while $\|\mathbf{r}^{(k)}\| > \mathsf{tol}\|\mathbf{b}\|$ **do**
  $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{z}^{(k)}$;
  $\mathbf{r}^{(k+1)} = \mathbf{b} - A\mathbf{x}^{(k+1)}$;
  $k = k + 1$;
end
$\mathbf{x} = \mathbf{x}^{(k)}$, res$= \|\mathbf{r}^{(k)}\|$, niter$= k$;

Algorithm 5.2 ensures that (5.9) is satisfied, but what can we say about the true error $\|\mathbf{e}^{(k)}\| = \|\mathbf{x} - \mathbf{x}^{(k)}\|$?

Note that the exact solution satisfies $A\mathbf{x} = \mathbf{b}$, and the approximate solution $\mathbf{x}^{(k)}$ satisfies

$$A\mathbf{x}^{(k)} = \mathbf{b} - \mathbf{r}^{(k)},$$

that is, the right-hand side is modified by $\mathbf{r}^{(k)}$. This matches the situation covered in Section 4.6 and from (4.6) we obtain that

$$\frac{\|\mathbf{x} - \mathbf{x}^{(k)}\|}{\|\mathbf{x}\|} \leq \kappa(A) \frac{\|\mathbf{b} - \hat{\mathbf{b}}\|}{\|\mathbf{b}\|} = \kappa(A) \frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{b}\|} \leq \kappa(A) \cdot \mathsf{tol},$$

where we recall that $\kappa(A)$ denotes the condition number of A. Hence, for moderate condition numbers, the error is reliably estimated by the residual. For high condition numbers, the residual may not provide a good estimate of the true error $\|\mathbf{x} - \mathbf{x}^{(k)}\|$.

5.5 Gradient methods

An important class of matrices which appears often in physics and engineering applications are symmetric positive definite (spd) matrices For example, such matrices arise when we are looking for an equilibrium configuration of a physical system that minimize its energy. We recall that a matrix $A \in \mathbb{R}^{n \times n}$ is **positive definite** if

$$\mathbf{v}^\top A\mathbf{v} > 0 \qquad \forall \mathbf{v} \in \mathbb{R}^n, \ \mathbf{v} \neq \mathbf{0}.$$

As mentioned before, a symmetric matrix is positive definite if and only if all its eigenvalues are positive.

A linear system $A\mathbf{x} = \mathbf{b}$ with spd A can be associated with an *energy function* ϕ defined as

$$\phi : \mathbb{R}^n \to \mathbb{R}, \qquad \phi(\mathbf{v}) = \frac{1}{2}\mathbf{v}^\top A\mathbf{v} - \mathbf{v}^\top \mathbf{b}, \qquad \mathbf{v} \in \mathbb{R}^n. \tag{5.10}$$

We have the following important characterization of the solution:

Proposition 5.3. *The solution* $\mathbf{x}$ *of* $A\mathbf{x} = \mathbf{b}$ *for an spd matrix* A *is the unique minimum of the function* ϕ*, that is,*

$$\mathbf{x} = \operatorname*{argmin}_{\mathbf{v}\in\mathbb{R}^n} \phi(\mathbf{v}).$$

We now discuss the case $n = 2$ in more detail.

Example 5.1. *Consider the linear system*

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x \\ y \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}_{\mathbf{b}}.$$

Then the energy function is

$$\begin{aligned} \phi(x, y) &= \frac{1}{2} \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ &= \frac{1}{2}a_{11}x^2 + a_{12}xy + \frac{1}{2}a_{22}y^2 - b_1x - b_2y. \end{aligned}$$

Its gradient is given by

$$\nabla\phi(x, y) = \begin{bmatrix} \frac{\partial\phi}{\partial x} \\ \frac{\partial\phi}{\partial y} \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y - b_1 \\ a_{12}x + a_{22}y - b_2 \end{bmatrix} = A\mathbf{x} - \mathbf{b}\,,$$

and the Hessian matrix is

$$H_\phi(x, y) = \begin{bmatrix} \frac{\partial^2\phi}{\partial x^2} & \frac{\partial^2\phi}{\partial x\partial y} \\ \frac{\partial^2\phi}{\partial x\partial y} & \frac{\partial^2\phi}{\partial y^2} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} = A.$$

The stationary points of ϕ *satisfy the condition*

$$\phi(x, y) = A\mathbf{x} - \mathbf{b} = \mathbf{0}.$$

As the system $A\mathbf{x} = \mathbf{b}$ *has only one solution, we conclude that there is exactly one stationary point, which coincides with the solution of the linear system. Moreover, as the Hessian matrix* $H_\phi = A$ *is spd, this stationary point is the unique minimum of the function* ϕ*.*

The calculations of Example 5.1 for $n = 2$ are valid for general n. In particular, we have that

$$\text{Gradient :} \qquad \nabla\phi(\mathbf{x}) = A\mathbf{x} - \mathbf{b}, \qquad \forall \mathbf{x} \in \mathbb{R}^n\,, \tag{5.11}$$

$$\text{Hessian :} \qquad H_\phi(\mathbf{x}) = A, \qquad \forall \mathbf{x} \in \mathbb{R}^n\,. \tag{5.12}$$

Thanks to this interpretation of the solution of a linear system as the minimum of an energy function, we can build iterative methods by trying to approximate the minimum of ϕ.

5.5.1 Gradient method (or steepest descent)

Gradient methods are widely used in optimization and relatively simple to implement. Here, we will only discuss the special case of a gradient method applied to to the energy function ϕ defined above.

The idea of the gradient method is relatively simple. Suppose we have an approximation $\mathbf{x}^{(k)}$ of the solution to $A\mathbf{x} = \mathbf{b}$, which is also an approximation of the minimum of ϕ. We aim at building a better approximation $\mathbf{x}^{(k+1)}$ for which $\phi(\mathbf{x}^{(k+1)}) < \phi(\mathbf{x}^{(k)})$. To this end, we are following the direction of the steepest slope of the function ϕ, trying to arrive as fast as possible at the minimum of ϕ. From (5.11), we know that the gradient of ϕ at $\mathbf{x}^{(k)}$ is

$$\nabla\phi(\mathbf{x}^{(k)}) = A\mathbf{x}^{(k)} - \mathbf{b} = -\mathbf{r}^{(k)}.$$

Therefore, the residual $\mathbf{r}^{(k)}$ gives the direction of steepest descent. We then define

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)} \tag{5.13}$$

for some steplength $\alpha_k > 0$ that will be determined below. Note that $\alpha_k = 1$ corresponds to the Richardson method with $P = I_n$.

To choose the steplength α_k, we use a technique called "exact line search", that is, we choose α_k such that $\phi(\mathbf{x}^{(k+1)})$ is as small as possible. Thanks to the simple form of ϕ, it is possible to determine an explicit expression for such α_k. For this purpose, we use the chain rule to compute

$$\begin{aligned}
\frac{\partial}{\partial\alpha}\phi(\mathbf{x}^{(k)} + \alpha\mathbf{r}^{(k)}) &= \nabla\phi\big(\mathbf{x}^{(k)} + \alpha\mathbf{r}^{(k)}\big)^\top \Big(\frac{\partial}{\partial\alpha}(\mathbf{x}^{(k)} + \alpha\mathbf{r}^{(k)})\Big) \\
&= \big(A(\mathbf{x}^{(k)} + \alpha\mathbf{r}^{(k)}) - \mathbf{b}\big)^\top \mathbf{r}^{(k)} \\
&= -\big(\mathbf{r}^{(k)}\big)^\top \mathbf{r}^{(k)} + \alpha\big(\mathbf{r}^{(k)}\big)^\top A\mathbf{r}^{(k)}.
\end{aligned}$$

For a minimum α_k, this derivative needs to zero and, hence, it follows that which gives us the optimum value

$$\alpha_k = \frac{(\mathbf{r}^{(k)})^\top \mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^\top A\mathbf{r}^{(k)}}.$$

Once the new approximation $\mathbf{x}^{(k+1)}$ is computed according to (5.13), we can update the residual using the formula

$$\mathbf{r}^{(k+1)} = \mathbf{b} - A\mathbf{x}^{(k+1)} = \mathbf{b} - A(\mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}) = \mathbf{r}^{(k)} - \alpha_k A\mathbf{r}^{(k)}.$$

The following algorithm implements the described gradient method, again using the stopping criterion based on the normalized residual.

Algorithm 5.3: Gradient method

Data: A, $\mathbf{b}$, $\mathbf{x}^{(0)}$, tol
Result: $\mathbf{x}$, res, niter
$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$; $k = 0$;
while $\|\mathbf{r}^{(k)}\| > \mathsf{tol} \cdot \|\mathbf{b}\|$ **do**
  $\mathbf{w}^{(k)} = A\mathbf{r}^{(k)}$;
  $\alpha_k = \frac{(\mathbf{r}^{(k)})^\top \mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^\top \mathbf{w}^{(k)}}$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}$;
  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{w}^{(k)}$;
  $k = k + 1$;
end
$\mathbf{x} = \mathbf{x}^{(k)}$, res= $\|\mathbf{r}^{(k)}\|$, niter= k;

For the convergence of the gradient method, we have the following result.

Theorem 5.4. *The gradient method applied to $A\mathbf{x} = \mathbf{b}$ with spd A always converges to the exact solution for any initial vector $\mathbf{x}^{(0)}$. Moreover, the error bound*

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C \left(\frac{\kappa(A) - 1}{\kappa(A) + 1} \right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\|, \tag{5.14}$$

holds for some constant C.

5.5.2 Generalizations

In this section, we will briefly discuss some generalizations of the gradient method.

Preconditioned gradient method

According to Theorem 5.4, the convergence speed of the gradient method is linked to the factor $(\kappa(A)-1)/(\kappa(A)+1)$. For large $\kappa(A)$, this factor is very close to 1 and the convergence can be expected to be very slow. To speed up convergence, we can use *preconditioning*.

As for the Richardson method, we assume the availability of a preconditioner $P \in \mathbb{R}^{n \times n}$ for which it is easy to solve linear systems. We assume that P is spd and multiply its inverse to both sides to $A\mathbf{x} = \mathbf{b}$:

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}. \tag{5.15}$$

The preconditioned gradient method is the gradient method applied to (5.15). The convergence of this method is controlled by the condition number of $P^{-1}A$. While all eigenvalues of $P^{-1}A$ are still real and positive, this matrix is generally *not* symmetric. We therefore have to slightly adjust the notion of condition number and set

$$\tilde{\kappa}(P^{-1}A) = \lambda_{\max}(P^{-1}A)/\lambda_{\min}(P^{-1}A).$$

One can then show that the iterates $\mathbf{x}^{(k)}$ of the preconditioned gradient method satisfy

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C \left(\frac{\tilde{\kappa}(P^{-1}A) - 1}{\tilde{\kappa}(P^{-1}A) + 1} \right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\|. \tag{5.16}$$

Thus, we will have fast convergence when $\tilde{\kappa}(P^{-1}A)$ remains modest.

The k^{th} iteration of the preconditioned gradient method computes the next iterate $\mathbf{x}^{(k+1)}$ as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)},$$

where $\mathbf{z}^{(k)}$ is the *preconditioned residual* (residual of the preconditioned system (5.15)) defined as

$$\mathbf{z}^{(k)} = P^{-1}\mathbf{b} - P^{-1}A\mathbf{x}^{(k)} = P^{-1}\mathbf{r}^{(k)},$$

and α_k is given by $\alpha_k = \frac{(\mathbf{z}^{(k)})^\top \mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^\top A \mathbf{z}^{(k)}}$.

Since we do not want to explicitly compute the inverse matrix P^{-1}, the update of the solution proceeds as follows:

$$\begin{aligned} \text{Solve} P\mathbf{z}^{(k)} &= \mathbf{r}^{(k)} \\ \text{Set} \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}. \end{aligned} \tag{5.17}$$

This results in the following algorithm:

Algorithm 5.4: Preconditioned gradient method

Given $\mathbf{x}^{(0)}$ and spd $P \in \mathbb{R}^{n\times n}$, set $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$;
for $k = 0, 1, \ldots$ **do**
  Solve $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$;
  $\mathbf{w}^{(k)} = A\mathbf{z}^{(k)}$;
  $\alpha_k = \frac{(\mathbf{z}^{(k)})^\top \mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^\top \mathbf{w}^{(k)}}$;
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$;
  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{w}^{(k)}$;
end

The choice of the matrix P is delicate. the On one hand, it has to be chosen such that $\tilde{\kappa}(P^{-1}A) \ll \kappa(A)$. On the other hand, the linear system

(5.17) has to be easily solvable. If we only considered the first criterion, the *ideal* choice of preconditioner would be $P = A$. In fact, in this case we have $\tilde{\kappa}(P^{-1}A) = \kappa(I) = 1$ and the method converges in only one iteration. However, with this choice, the linear system (5.17) to solve in every iteration is as complicated as the initial system and we do not gain anything. Therefore, we have to find a good compromise: the matrix P must be close to A in some sense, while keeping the solution of the linear system (5.17) relatively simple.

One possibility already considered in the context of the Jacobi method is to construct P from the diagonal of A.

In the case of a sparse matrix, another very common choice is to compute an incomplete LU decomposition of the matrix A, that is, $P = \hat{L}\hat{U}$ where $\hat{L}$ and $\hat{U}$ are approximations of the factors L and U of A. This technique is called ILU (*incomplete* LU). In particular, we can perform an LU decomposition of A and save only the "larger" entries of the matrices L and U (ILUt: incomplete LU with threshold). An even more radical choice is to not allow for any fill-in and only save the entries at positions where A is nonzero (ILU0). In `Python`, an ILU decomposition is computed using scipy.sparse.linalg.spilu .

ILU avoids all problems related to fill-in and (hopefully) gives a matrix $P = \hat{L}\hat{U}$ that is close enough to A. Note that the system $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$ is easy to solve because the P matrix is already factored.

Conjugate gradient method

The gradient method takes the form

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}$$

where $\mathbf{r}^{(k)} = -\nabla\phi(\mathbf{x}^{(k)})$ is the direction of steepest descent of ϕ.

It is natural to ask whether there are other descent directions that allows one to reach the minimum of ϕ in less iterations. A strategy that works particularly well consists in choosing at iteration k a direction $\mathbf{p}^{(k)}$, which has the following property:

$$(\mathbf{p}^{(k)})^\top A\mathbf{p}^{(j)} = 0, \qquad j = 0, \ldots, k-1. \tag{5.18}$$

Vectors $\mathbf{p}^{(k)}$ satisfying (5.18) are called *A-conjugate* or *A-orthogonal*. In this case, the update of the solution is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$$

where

$$\alpha_k = \frac{(\mathbf{p}^{(k)})^\top \mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^\top A\mathbf{p}^{(k)}}.$$

The direction $\mathbf{p}^{(k)}$ can be computed with the following recurrence:

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)}, \quad \beta_k = \frac{(A\mathbf{p}^{(k)})^\top \mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^\top A\mathbf{p}^{(k)}} \quad \text{and} \quad \mathbf{p}^{(0)} = \mathbf{r}^{(0)}.$$

The described method is called conjugate gradient method, which has the following convergence property.

Theorem 5.5. *The conjugate gradient method applied to the linear system $A\mathbf{x} = \mathbf{b}$, with spd A, converges in at most n iterations to the exact solution for any initial datum $\mathbf{x}^{(0)}$ (in exact arithmetic). Moreover, the error at the k^{th} iteration satisfies the bound the following estimation*

$$\|\mathbf{x} - \mathbf{x}^{(k)}\| \leq C \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|\mathbf{x} - \mathbf{x}^{(0)}\|, \tag{5.19}$$

for some positive constant C.

We notice that the conjugate gradient method converges faster than the gradient method as the reduction factor of the error is controlled by $\sqrt{\kappa(A)}$ instead of $\kappa(A)$.

Again, we can combine the conjugate gradient method with preconditioning, which leads to PCG (*preconditioned conjugate gradient*). With a good choice of preconditioner, the PCG method is very effective for the solution of linear systems. In fact, PCG is *the* iterative method of choice for solving large-scale spd linear systems. In `Python`, PCG is implemented in scipy.sparse.linalg.cg .

Chapter 6

Ordinary differential equations

In this chapter we are interested in the numerical solution of an ordinary differential equation (ODE) of the form: *Find a continuously differentiable function* $u : \mathbb{R}_+ \to \mathbb{R}$ *such that*

$$\begin{cases} \dfrac{du(t)}{dt} = f(t, u(t)), \quad t > 0, \\ u(0) = u_0, \end{cases} . \tag{6.1}$$

This problem is called a *Cauchy problem.*

In applications, we often encounter *systems of ordinary differential equations.* Given

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ \vdots \\ u_m(t) \end{bmatrix} \qquad \text{and} \qquad \mathbf{f}(t, \mathbf{u}) = \begin{bmatrix} f_1(t, u_1, \ldots, u_m) \\ \vdots \\ f_m(t, u_1, \ldots, u_m) \end{bmatrix} ,$$

we are looking for a vector-valued function $\mathbf{u} : \mathbb{R}_+ \to \mathbb{R}^m$ *such that*

$$\begin{cases} \dfrac{d\mathbf{u}(t)}{dt} = \mathbf{f}(t, \mathbf{u}(t)), \quad t > 0 \\ \mathbf{u}(0) = \mathbf{u}_0 \end{cases} . \tag{6.2}$$

Example 6.1. *Let us derive a model for the evolution of a rabbit population. Let* $u(t)$ *be the number of individuals at any time* t *and* $u_0 = u(0)$ *the number of individuals at the initial time* $t = 0$*. It is reasonable to assume that the growth rate of the population at any time* t*, that is, the number of individuals being born minus the number of individuals deceased in one unit of time, is proportional to the total number* $u(t)$ *of individuals in the population:*

$$\textit{growth rate at any given time } t: \qquad \tau(t) = Cu(t).$$

Thus, we can model the population dynamics using the following ODE:

$$\begin{cases} \dfrac{du(t)}{dt} = Cu(t), \quad t > 0, \\ u(0) = u_0. \end{cases} \tag{6.3}$$

The solution of (6.3) *is* $u(t) = u_0 e^{Ct}$. *This shows that the model is realistic; it foresees that the population will increase indefinitely with time! As the food available for the rabbits is not infinite, a more realistic model predicts that the number of individuals cannot exceed a certain value* $u_{\max}$. *Thus, another possible differential model is*

$$\begin{cases} \dfrac{du(t)}{dt} = Cu(t)\left(1 - \dfrac{u(t)}{u_{\max}}\right), \quad t > 0, \\ u(0) = u_0. \end{cases} . \tag{6.4}$$

In this model, the growth rate $\tau(t) = Cu(t)\left(1 - \frac{u(t)}{u_{\max}}\right)$ *becomes zero when the population reaches the maximum value* $u_{\max}$ *and the population cannot grow anymore.*

Example 6.2. *We now consider two populations: rabbits and foxes. Let* $u_1(t)$ *be the number of rabbits and* $u_2(t)$ *the number of foxes at any given time* t. *For each of these populations individually, we can use an equation of the type* (6.3) *or* (6.4). *Let simplicity, we use* (6.3). *This time, however, the dynamics of the two populations are* linked. *In fact, foxes eat rabbits and therefore the food available for the foxes depends on the number of rabbits* $u_1(t)$. *Moreover, the mortality rate of rabbits depends on the number of foxes. We can then write the following model*

$$\begin{cases} \dfrac{du_1(t)}{dt} = \alpha u_1(t) - \beta u_1(t) u_2(t), \quad t > 0, \\ \dfrac{du_2(t)}{dt} = -\gamma u_2(t) + \delta u_1(t) u_2(t), \quad t > 0, \\ u_1(0) = u_{1,0}, \qquad u_2(0) = u_{2,0}, \end{cases}$$

where $\alpha u_1(t)$ *is the number of birth minus the natural death rate of the rabbit population;* $-\beta u_1(t) u_2(t)$ *is the mortality rate of rabbits due to the presence of foxes;* $-\gamma u_2(t)$ *is the natural death rate of foxes;* $\delta u_1(t) u_2(t)$ *is the birth rate of foxes, which is proportional to the number of rabbits alive. This model is known as* Lotka-Volterra *model. It is a system composed of two coupled ODEs, which can be written in the vector form* (6.2) *with*

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}, \qquad \mathbf{f}(t, \mathbf{u}(t)) = \begin{bmatrix} \alpha u_1(t) - \beta u_1(t) u_2(t) \\ -\gamma u_2(t) + \delta u_1(t) u_2(t) \end{bmatrix}.$$

6.1 Existence and uniqueness of solutions

In this section, we briefly recall results on the existence and uniqueness of solutions to ODEs. Both, existence and uniqueness, are nontrivial questions, as shown by the following two examples.

Example 6.3. *Let us consider the Cauchy problem*

$$\frac{du(t)}{dt} = \sqrt{u(t)}, \;\; t > 0, \qquad u(0) = 0.$$

It is easy to check that $u_1(t) = 0, \; t > 0$ *and* $u_2(t) = \frac{1}{4}t^2, \; t > 0$ *are two possible solutions and, hence, this Cauchy problem does not have a unique solution.*

Example 6.4. *Let us consider the Cauchy problem*

$$\frac{du(t)}{dt} = u^2(t), \;\; t > 0, \qquad u(0) = 1.$$

The solution is $u(t) = \frac{1}{1-t}$*, which only exists for* $t < 1$*.*

In our study of methods for the numerical solution of ODEs, we want to exclude the situations encountered in Examples 6.3 and 6.4. For this purpose, we recall a result on the existence and uniqueness of the solution of (6.1) for all $t > 0$.

Theorem 6.1. *Suppose that* $f : \mathbb{R} \times \mathbb{R}_+ \to \mathbb{R}$ *is with respect to both arguments and* Lipschitz-continuous *with respect to its second argument, that is, there exists a constant* $L > 0$ *(the Lipschitz constant) such that*

$$|f(t,x) - f(t,y)| \leq L|x - y|, \qquad \forall x, y \in \mathbb{R}, \; \forall t \in [0, \infty).$$

Then the Cauchy problem (6.1) *has a unique solution* $u(t)$ *defined for all* $t \in [0, \infty)$*. Moreover, the solution is continuously differentiable.*

Theorem 6.1 generalizes to systems of ODEs. Throughout the rest of this chapter, we will assume that the hypotheses of Theorem 6.1 are verified.

6.2 One-step methods

We aim at approximating the solution to the Cauchy problem (6.1) in the interval $[0, T]$ for a fixed end time $T > 0$. For this purpose, we divide the interval in N sub-intervals $[t_n, t_{n+1}]$ of the same length $\Delta t = \frac{T}{N}$, such that $t_n = n\Delta t$, $n = 0, ..., N$. The methods discussed in this chapter approximate the solution, starting from t_0, subsequently interval by interval.

Forward Euler method (also called Explicit Euler)

A simple idea to approximate the solution on the interval $[t_n, t_{n+1}]$ consists in replacing the derivative $\frac{du}{dt}$ at time t_n with the forward finite differences formula:

$$\frac{du(t_n)}{dt} \approx \delta^+_{\Delta t} u(t_n) = \frac{u(t_{n+1}) - u(t_n)}{\Delta t}.$$

Letting $u^n \approx u(t_n)$ denote the approximation of $u(t_n)$, we obtain an approximation $u^{n+1} \approx u(t_{n+1})$ by performing one-step of the so called **forward Euler method**:

$$\begin{cases} \dfrac{u^{n+1} - u^n}{\Delta t} = f(t_n, u^n), \quad n = 0, 1, \ldots, N-1 \\ u^0 = u_0 \ \text{ (known)} \end{cases}. \tag{6.5}$$

For $n = 0$, this means that u^1 is computed from u^0 (which is known) as

$$u^1 = u^0 + \Delta t\, f(t_0, u^0).$$

Once u^1 is computed, by substituting $n = 1$ in (6.5) we can find the value of u^2 and so on. The explicit Euler method produces a sequence $(u^0, u^1, u^2, \ldots, u^N)$ which we expect to be a good approximation of the exact solution at the corresponding time steps $(u_0 = u(t_0), u(t_1), u(t_2), \ldots, u(t_N) = u(T))$. The term *explicit* refers to the fact that we can compute the solution u^{n+1} *explicitly* from the solution u^n.

A `Python` implementation of the forward Euler method:

```
import numpy as np

def euler(f, I, u0, N):
    #    Solves the Cauchy problem
    #       u'=f(t,u), t in (t0,T], u(t0)=u0
    #    using the forward Euler method with a time step dt=(T-t0)/N
    #    Input:
    #      f: function f(t, u)
    #      I: the integration interval [t0,T]
    #      u0: initiale condition
    #      N: number of subintervals
    #    Output:
    #      t: vector of time instants tn
    #      u: approximate solution un
    #      dt: time step
    dt = (I[1] - I[0]) / N
    t = np.linspace(I[0], I[1], N + 1)
    u = np.zeros(N + 1)
    u[0] = u0
    for n in range(N):
        u[n + 1] = u[n] + dt * f(t[n], u[n])
    return t, u, dt
```

Example 6.5. *We apply the forward Euler method to the Cauchy problem*

$$\begin{cases} \frac{du(t)}{dt} = -(\frac{1}{2}u + 3te^{-t}), \quad t \in (0, 20], \\ u(0) = 1. \end{cases} \tag{6.6}$$

The exact solution is given by $u_{\text{ex}} = -11e^{-\frac{1}{2}t} + 12(1 + \frac{1}{2}t)e^{-t}$. *The following code computes the approximate solution using the Euler method with* $N = 20$ $(\Delta t = 1)$ *and compares it with the exact solution:*

```
import numpy as np
import matplotlib.pyplot as plt
from ch6_euler import euler

a = 0.5
b = 3
f = lambda t, u: -(a * u + b * t * np.exp(-t))
u0 = 1
Tf = 20
I = [0, Tf]
uex = (
    lambda t: (u0 - b / (1 - a) ** 2) * np.exp(-a * t)
    + b * (1 + (1 - a) * t) * np.exp(-t) / (1 - a) ** 2
)
t = np.linspace(0, Tf, 2001)
plt.plot(t, uex(t), "r")
N = 20
tn, un, dt = euler(f, I, u0, N)
plt.plot(tn, un, "g*-")
```

Figure 6.1 shows the obtained result as well as the approximations obtained for $N = 40$ $(\Delta t = 0.5)$ *and* $N = 80$ $(\Delta t = 0.25)$. *The forward Euler*

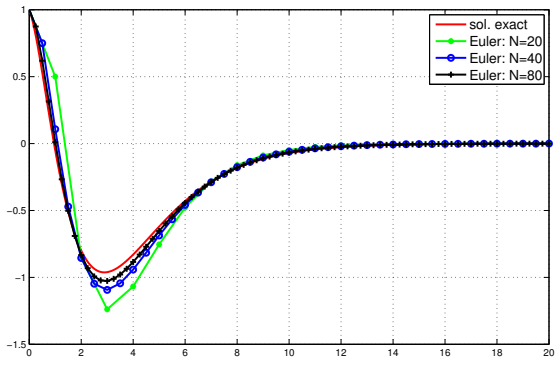


Figure 6.1: Exact solution of Cauchy problem (6.6) and approximations obtained with forward Euler with $N = 20, 40, 80$.

method is seen to yield approximated solutions that become increasingly accurate when the number N of sub-intervals increases or, equivalently, when the time step Δt decreases.

Backward Euler method (also called Implicit Euler)

Instead of the forward finite difference for approximating $\frac{du(t_n)}{dt}$, one can consider other choices, such as the backward finite difference:

$$\frac{du(t_n)}{dt} \approx \delta^-_{\Delta t} u(t_n) = \frac{u(t_n) - u(t_{n-1})}{\Delta t}.$$

This choice results in the **backward Euler method**

$$\begin{cases} \dfrac{u^{n+1} - u^n}{\Delta t} = f(t_{n+1}, u^{n+1}), \quad n = 0, 1, \ldots, N-1 \\ u^0 = u_0 \ \text{(known)}. \end{cases} \tag{6.7}$$

Given u^n, we now need to solve the equation

$$u^{n+1} - \Delta t\, f(t_{n+1}, u^{n+1}) = u^n$$

to determine the next approximation u^{n+1}. This is a non-linear equation in the unknown u^{n+1}. Setting $x = u^{n+1}$, we thus have to find the root of the function

$$g_n(x) = x - \Delta t\, f(t_{n+1}, x) - u^n = 0. \tag{6.8}$$

This implicit definition of u^{n+1} explains the name *implicit* of the method. For solving (6.8), we can use one of the methods introduced in Chapter 1. For example, one could use the following fixed-point method:

$$x^{(k+1)} = \Delta t\, f(t_{n+1}, x^{(k)}) + u^n.$$

Since we expect the solution u^{n+1} to be close to u^n, we can use $x^{(0)} = u^n$ as initial value for the fixed-point method. The Newton method

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - \Delta t\, f(t_{n+1}, x^{(k)}) - u^n}{1 - \Delta t \frac{\partial f}{\partial x}(t_{n+1}, x^{(k)})}, \quad k = 0, 1, \ldots, \qquad x^{(0)} = u^n,$$

can be expected to converge faster. However, there is no need compute the solution more accurately than the approximation error introduced by the finite difference approximation. In fact, one-step of the Newton method is usually sufficient.

At first glance, the backward Euler method seems to offer little benefit and in view of the additional complications implied by need for solving a nonlinear equation in every time step. However, we will see in Section 6.4 that this method has better stability properties compared to the forward Euler method.

Crank-Nicolson method

Another way to approximate the solution of (6.1) is obtained via quadrature. For this purpose, we integrate (6.1) between t_n and t_{n+1},

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} \frac{du(s)}{ds}\,\mathrm{d}s = \int_{t_n}^{t_{n+1}} f(s, u(s))\,\mathrm{d}s,$$

and then apply the trapezoidal formula to approximate the integral:

$$\int_{t_n}^{t_{n+1}} f(s, u(s))\,\mathrm{d}s \approx \frac{\Delta t}{2}\left[f(t_n, u(t_n)) + f(t_{n+1}, u(t_{n+1}))\right].$$

This gives us the **Crank-Nicolson method**:

$$\begin{cases} \dfrac{u^{n+1} - u^n}{\Delta t} = \dfrac{1}{2}f(t_n, u^n) + \dfrac{1}{2}f(t_{n+1}, u^{n+1}), \quad n = 0, 1, \ldots, N-1, \\ u^0 = u_0 \ \text{(known)}. \end{cases} \tag{6.9}$$

Once again this an *implicit* method, which requires the solution of a nonlinear equation in every time step. We will see that this scheme is usually more accurate than Euler's method (forward and backward)

Heun method

To avoid the need for solving a nonlinear equation in the Crank-Nicolson method, one can first compute an approximation of u^{n+1} using forward Euler:

$$\tilde{u}^{n+1} = u^n + \Delta t\, f(t_n, u^n).$$

Using $\tilde{u}^{n+1}$ instead of u^{n+1} in the right-hand side of (6.9) gives

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}f(t_n, u^n) + \frac{1}{2}f(t_{n+1}, \tilde{u}^{n+1}).$$

Combining both equations results in the **Heun method**

$$\begin{cases} \dfrac{u^{n+1} - u^n}{\Delta t} = \dfrac{1}{2}f(t_n, u^n) + \dfrac{1}{2}f(t_{n+1}, u^n + \Delta t\, f(t_n, u^n)), \\ u^0 = u_0 \ \text{(known)}, \end{cases} \tag{6.10}$$

with $n = 0, 1, \ldots, N-1$. Note that this method is *explicit* and as we can compute the new solution u^{n+1} in an explicit way from u^n using

$$u^{n+1} = u^n + \frac{\Delta t}{2}f(t_n, u^n) + \frac{\Delta t}{2}f(t_{n+1}, u^n + \Delta t\, f(t_n, u^n)).$$

Up to now, every method we studied computes u^{n+1} using the last step u^n only. To reflect this property, we call them *one step methods*.

Definition 6.1. *A **one-step method** to approximate solution to the Cauchy problem* (6.1) *is a method of the form*

$$\frac{u^{n+1} - u^n}{\Delta t} = \phi_f(u^n, u^{n+1}, t_n, \Delta t). \tag{6.11}$$

If the function ϕ_f does not depend on u^{n+1}, the method is called explicit, and otherwise implicit.

The following table collects the functions ϕ_f for the methods studied above:

Method	$\phi_f(u^n, u^{n+1}, t_n, \Delta t) =$
Forward Euler	$f(t_n, u^n)$
Backward Euler	$f(t_n + \Delta t, u^{n+1})$
Crank Nicolson	$\frac{1}{2}f(t_n, u^n) + \frac{1}{2}f(t_n + \Delta t, u^{n+1})$
Heun	$\frac{1}{2}f(t_n, u^n) + \frac{1}{2}f(t_n + \Delta t, u^n + \Delta t\, f(t_n, u^n))$

Multistep methods construct solution u^{n+1} by not only using u^n but also u^{n-1}, u^{n-2}, etc. Such methods will not be discussed in this lecture.

6.3 Error analysis

Given a one-step method of the form (6.11), we now aim at studying the behavior of the approximation error. Specifically, we consider the error at the final step T:

$$\varepsilon_N = |u(T) - u^N|, \quad \text{where} \quad N = T/\Delta t.$$

We have already seen for the forward Euler method that the error decreases as the time step size Δt decreases. The following definitions captures the quality of this decrease.

Definition 6.2. *A numerical method that performs N time steps of size Δt to approximate the Cauchy problem* (6.1) *is said to be **convergent of order** p if there is a constant $C > 0$ such that*

$$|u(T) - u^N| \leq C\Delta t^p,$$

provided that the exact solution is sufficiently smooth.

To study the convergence order of a numerical method, we have to introduce some additional concepts.

Definition 6.3 (**Local truncation error**)**.** *Let $u(t_n)$ be the exact solution of the Cauchy problem* (6.1) *at time $t_n = n\Delta t$ for $n = 0, 1, \ldots, N$. The* local truncation error *of a one-step method* (6.11) *at instant t_{n+1} is defined as*

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - \phi_f(u(t_n), u(t_{n+1}), t_n, \Delta t). \tag{6.12}$$

The overall truncation error is defined as

$$\tau = \max_{n=0,1,\ldots,N-1} |\tau_n|.$$

Let us emphasize that the truncation error is the error committed by the one-step method at t_{n+1} *when the previous approximation* u_n *is replaced by the exact value* $u(t_n)$. Of course, this exact value is not known unless $n = 0$, so the actual error of the one-step method at t_{n+1} is a combination of the truncation error τ_n and the errors resulting from the first n time steps of the method.

Example 6.6 (Truncation error of the forward Euler method)**.** *For the forward Euler method we have*

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - f(t_n, u(t_n)).$$

We recall that the approximation error of forward finite differences (see Chapter 3) satisfies

$$\left| u'(t_n) - \frac{u(t_{n+1}) - u(t_n)}{\Delta t} \right| \leq \frac{\Delta t}{2} \max_{t\in[t_n,t_{n+1}]} |u''(t)|.$$

As $u(t)$ *is the* exact *solution of the Cauchy problem* (6.1)*, it satisfies* $u'(t_n) = f(t_n, u(t_n))$ *and thus*

$$|\tau_n| = \left| \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - u'(t_n) \right| \leq \frac{\Delta t}{2} \max_{t\in[t_n,t_{n+1}]} |u''(t)|.$$

Therefore,

$$\tau \leq C\Delta t \quad \textit{with} \quad C = \frac{1}{2} \max_{t\in[0,T]} |u''(t)|.$$

Example 6.7 (Truncation error of the Crank-Nicolson method)**.** *For the Crank-Nicolson method, we have*

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - \frac{1}{2} f(t_n, u(t_n)) - \frac{1}{2} f(t_{n+1}, u(t_{n+1})).$$

We recall that the error of the trapezoidal quadrature formula on a single interval (see Chapter 3) satisfies

$$\left| \int_{t_n}^{t_{n+1}} g(s)ds - \frac{\Delta t}{2} \left(g(t_n) + g(t_{n+1}) \right) \right| \leq C \max_{t\in[t_n,t_{n+1}]} |g''(t)| \Delta t^3.$$

Inserting $g(t) = u'(t) = f(t, u(t))$ *allows us to write*

$$|\tau_n| = \frac{1}{\Delta t} \left| \int_{t_n}^{t_{n+1}} u'(s)ds - \frac{\Delta t}{2} \left[f(t_n, u(t_n)) + f(t_{n+1}, u(t_{n+1})) \right] \right| \leq C \max_{t\in[t_n,t_{n+1}]} |u'''(t)| \Delta t^2,$$

and thus

$$\tau \leq C \max_{t\in[0,T]} |u'''(t)| \Delta t^2.$$

In general, one has the following result for one-step methods: *If the truncation error is of order p, that is, $\tau \leq C\Delta t^p$, then the numerical method is convergent of order p.*

We will prove this result for the forward Euler method only.

Proof for forward Euler method. Let us set

$$\tilde{u}^{n+1} = u(t_n) + \Delta t f(t_n, u(t_n)), \qquad n = 0, \ldots, N-1,$$

which allows us to express the local truncation error as

$$\tau_n = \frac{u(t_{n+1}) - \tilde{u}^{n+1}}{\Delta t}.$$

In order to bound the total error $u(t_{n+1}) - u^{n+1}$, we split it into two parts:

$$\begin{aligned}
|u(t_{n+1}) - u^{n+1}| &\leq \underbrace{|u(t_{n+1}) - \tilde{u}^{n+1}|}_{=\Delta t|\tau_n|} + |\tilde{u}^{n+1} - u^{n+1}| \\
&= \Delta t|\tau_n| + |u(t_n) - u^n + \Delta t(f(t_n, u(t_n)) - f(t_n, u^n))| \\
&\leq \Delta t|\tau_n| + |u(t_n) - u^n| + \Delta t \underbrace{|f(t_n, u(t_n)) - f(t_n, u^n)|}_{\leq L|u(t_n)-u^n| \text{ as } f \text{ is Lipschitz.}} \\
&\leq \Delta t\tau + (1 + L\Delta t)|u(t_n) - u^n|.
\end{aligned}$$

Defining $\varepsilon_n = |u(t_n) - u^n|$, the error at time t_n, applying the previous relation repeatedly allows us to link ε_{n+1} to ε_n, then to ε_{n-1}, and so on, until arriving at $\varepsilon_0 = |u(t_0) - u^0| = 0$:

$$\begin{aligned}
\varepsilon_{n+1} &\leq \Delta t\tau + (1 + L\Delta t)\varepsilon_n \\
&\leq \Delta t\tau + (1 + L\Delta t)\Delta t\tau + (1 + L\Delta t)^2\varepsilon_{n-1} \\
&\vdots \\
&\leq \sum_{i=0}^{n} (1 + L\Delta t)^i \Delta t\tau + (1 + L\Delta t)^{n+1}\varepsilon_0 \\
&= \frac{(1 + L\Delta t)^{n+1} - 1}{L}\tau,
\end{aligned}$$

where the last step uses the identity

$$\sum_{i=0}^{n} r^k = \frac{1 - r^{n+1}}{1 - r}$$

for $r = 1 + L\Delta t > 1$. We now use the inequality $(1 + x) \leq e^x$, $x \geq 0$, to conclude

$$|u(T) - u^N| = \varepsilon_N \leq \frac{e^{L\Delta t N} - 1}{L}\tau = \frac{e^{LT} - 1}{L}\tau,$$

which proves that the error at the final step is of the same order as the truncation error. □

The order of the one-step methods discussed so far is summarized in the following table:

Method	order
Forward Euler	1
Backward Euler	1
Crank Nicolson	2
Heun	2

6.4 Absolute stability

An *autonomous* system of differential equations takes the form

$$\begin{cases} \dfrac{d\mathbf{u}(t)}{dt} = \mathbf{f}(\mathbf{u}(t)), \quad t > 0, \\ \mathbf{u}(0) = \mathbf{u}_0, \end{cases} \tag{6.13}$$

that is, the function $\mathbf{f}$ does not depend explicitly on time. The system of Example 6.2 is an instance of an autonomous system.

The values $\bar{\mathbf{u}}$ (if they exist) for which $\mathbf{f}(\bar{\mathbf{u}}) = \mathbf{0}$ are called *equilibrium points*. If we set $\mathbf{u}_0 = \bar{\mathbf{u}}$, the solution of (6.13) is $\mathbf{u}(t) = \bar{\mathbf{u}}$ for all t; the system remains in equilibrium.

An equilibrium point is called a *global attractor* if, for all $\mathbf{u}_0$, the solution of (6.13) satisfies $\lim_{t\to\infty} \mathbf{u}(t) = \bar{\mathbf{u}}$. In other words, any solution of (6.13) approaches the equilibrium after some (long) time. For many reasons, this is an important property and, ideally, the approximation produced by a numerical method should preserve it. This gives rise to the following *stability* question: *Given an autonomous system* (6.13) *with a global attractor* $\bar{\mathbf{u}}$ *such that*

$$\forall\, \mathbf{u}_0, \qquad \lim_{t\to\infty} \mathbf{u}(t) = \bar{\mathbf{u}},$$

is it true that the approximation $\mathbf{u}^n$ *computed by a numerical method also satisfies*

$$\forall\, \mathbf{u}_0, \qquad \lim_{n\to\infty} \mathbf{u}^n = \bar{\mathbf{u}}\ ?$$

In other words, if the exact solution approaches the equilibrium value, is the numerical solution also approaching this equilibrium value? As we will see, this is not always the case.

The answer to the stability question raised above is difficult to answer in the general nonlinear case. However, we can give precise answers for a *linear* system:

$$\frac{d\mathbf{u}(t)}{dt} = A\mathbf{u}(t) + \mathbf{b}, \ \ t > 0, \qquad \mathbf{u}(0) = \mathbf{u}_0, \tag{6.14}$$

where $\mathbf{u}(t) = (u_1(t), \ldots, u_m(t))^\top$, $A \in \mathbb{R}^{m\times m}$ and $\mathbf{b} \in \mathbb{R}^m$.

We have the following result regarding equilibrium points for (6.14).

Theorem 6.2. *Given a linear system* (6.14)*, suppose that* $\Re(\lambda_i(A)) < 0$ *for every eigenvalue* $\lambda_i(A) \in \mathbb{C}$ *of* A*. Then* (6.14) *has exactly one equilibrium point* $\bar{\mathbf{u}} = -A^{-1}\mathbf{b}$*, which is a global attractor:*

$$\forall\, \mathbf{u}_0, \qquad \lim_{t\to\infty} \mathbf{u}(t) = \bar{\mathbf{u}} = -A^{-1}\mathbf{b}.$$

We now investigate when the property of Theorem 6.2 is preserved by a numerical method.

Definition 6.4. *Under the hypothesis of Theorem 6.2:*

- *A numerical method that computes approximations* $\mathbf{u}^n \approx \mathbf{u}(t_n)$ *at uniform time steps is called* ***absolutely stable*** *for a fixed time step size* Δt *if*
$$\forall\, \mathbf{u}_0 \qquad \lim_{n\to\infty} \mathbf{u}^n = \bar{\mathbf{u}} = -A^{-1}\mathbf{b}.$$
- *A numerical method is called* ***unconditionally absolutely stable*** *(or A-stable) if it is stable for all* $\Delta t > 0$*. Otherwise, if the method is stable only for some* $\Delta t > 0$*, it is called* ***conditionally absolutely stable****.*

To verify whether a numerical method is absolutely stable, it suffices to consider the case $\mathbf{b} = 0$. Indeed, after performing the substitution $\mathbf{v}(t) = \mathbf{u}(t) - \bar{\mathbf{u}}$ for $\mathbf{u}(t)$ satisfying (6.14), we see that $\mathbf{v}(t)$ satisfies the homogeneous system $\frac{d\mathbf{v}(t)}{dt} = A\mathbf{v}(t)$ with the initial value $\mathbf{v}(0) = \mathbf{u}_0 - \bar{\mathbf{u}}$.

6.4.1 Scalar model problem

We will first study the scalar case as a warmup:

$$\frac{du(t)}{dt} = \lambda u(t), \;\; t > 0, \qquad u(0) = u_0. \tag{6.15}$$

We assume $\lambda < 0$, which implies $u(t) = u_0 e^{\lambda t} \to 0$ for $t \to \infty$.

Lemma 6.3. *The* forward Euler method *applied to* (6.15) *is absolutely stable if*

$$\Delta t < \frac{2}{|\lambda|}. \tag{6.16}$$

The backward Euler method *applied to* (6.15) *is unconditionally absolutely stable.*

Proof. The forward Euler method applied to (6.15) takes the form

$$u^n = u^{n-1} + \Delta t\lambda u^{n-1} = (1 + \Delta t\lambda)u^{n-1} = \ldots = (1 + \Delta t\lambda)^n u_0.$$

Therefore $u^n \xrightarrow{n\to\infty} 0$ if and only if $|1+\Delta t\lambda| < 1$, which gives the condition (6.16).

The backward Euler method applied to (6.15) takes the form

$$u^n = u^{n-1} + \Delta t\lambda u^n \qquad \Longrightarrow \qquad u^n = \frac{u^{n-1}}{1-\Delta t\lambda} = \ldots = \frac{1}{(1-\Delta t\lambda)^n} u_0.$$

As $\lambda < 0$, we have $|\frac{1}{1-\Delta t\lambda}| < 1$ and $u^n \xrightarrow{n\to\infty} 0$ for all $\Delta t > 0$, which proves the second part. □

6.4.2 Vector model problem

We now consider the homogeneous case of the linear system (6.14):

$$\frac{d\mathbf{u}(t)}{dt} = A\mathbf{u}(t), \;\; t > 0, \qquad \mathbf{u}(0) = \mathbf{u}_0, \tag{6.17}$$

and assume that $\Re(\lambda_i(A)) < 0$ for $i = 1, \ldots, m$.

The following lemma generalizes the result of Lemma 6.3.

Lemma 6.4. *The* forward Euler method *applied to* (6.17) *is absolutely stable if* $|1+\Delta t\lambda_i| < 1$ *holds for every eigenvalue* λ_i *of* A *or, equivalently,*

$$\Delta t < \min_{i=1,\ldots,m} \frac{2|\Re\lambda_i|}{|\lambda_i|^2}. \tag{6.18}$$

The backward Euler method applied to (6.17) *is unconditionally absolutely stable.*

Proof. To simplify the proof, we assume that A is diagonalizable (the result also holds for non-diagonalizable A). Hence, there is an invertible matrix V (containing the eigenvectors of A) such that

$$A = VDV^{-1}, \quad D = \operatorname{diag}(\lambda_1, \ldots, \lambda_n).$$

The forward Euler method takes the form

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = A\mathbf{u}^n = VDV^{-1}\mathbf{u}^n.$$

If we multiply both sides of this equation with V^{-1} and perform the change of variable $\mathbf{w}^n = V^{-1}\mathbf{u}^n$, we obtain that

$$\frac{\mathbf{w}^{n+1} - \mathbf{w}^n}{\Delta t} = V^{-1}VDV^{-1}\mathbf{u}^n = D\mathbf{w}^n.$$

Because D is diagonal, the i^{th} entry of this equation reads as

$$\frac{w_i^{n+1} - w_i^n}{\Delta t} = \lambda_i w_i^n \quad \Longrightarrow \quad w_i^{n+1} = (1+\lambda_i\Delta t) w_i^n, \;\; i = 1, \ldots, m.$$

Hence, it follows that every entry of the vector $\mathbf{w}^n$, and thus also the vector $\mathbf{u}^n = V\mathbf{w}^n$, converges to zero as $n \to \infty$ if and only if

$$|1 + \lambda_i \Delta t| < 1 \qquad \forall i = 1, \dots, m.$$

which corresponds to

$$(1 + \Delta t \Re(\lambda_i))^2 + \Delta t^2 \Im(\lambda_i)^2 < 1$$

$$\iff \quad 2\Re(\lambda_i) + \Delta t |\lambda_i|^2 < 0,$$

giving the condition (6.18) using that $\Re(\lambda_i) < 0$ and $\Delta t > 0$.

On the other hand, the backward Euler method takes the form

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = A\mathbf{u}^{n+1} = VDV^{-1}\mathbf{u}^{n+1}.$$

After again performing the change of variable $\mathbf{w}^n = V^{-1}\mathbf{u}^n$, we get

$$\frac{\mathbf{w}^{n+1} - \mathbf{w}^n}{\Delta t} = D\mathbf{w}^{n+1},$$

which entrywise reads as

$$\frac{w_i^{n+1} - w_i^n}{\Delta t} = \lambda_i w_i^{n+1} \quad \implies \quad w_i^{n+1} = \frac{1}{1 - \lambda_i \Delta t} w_i^n, \;\; i = 1, \dots, m.$$

Since $\Re(\lambda_i) < 0$ for every $i = 1, \dots, m$, it follows that

$$|w_i^{n+1}| = \frac{|w_i^n|}{|1 - \lambda_i \Delta t|} = \frac{|w_i^n|}{\sqrt{(1 - \Re(\lambda_i)\Delta t)^2 + \Im(\lambda_i)^2 \Delta t^2}} < |w_i^n|.$$

Thus, $\mathbf{w}^n = V^{-1}\mathbf{u}^n \to 0$ for $n \to \infty$ for all $\Delta t > 0$, which proves that the method is A-stable. □

Lemma 6.4 shows that forward Euler is only stable for sufficiently small step sizes. It can be shown that *every* explicit method requires such a step size restriction.

6.5 Error control: An adaptive algorithm

So far, we have assumed that the time step size is constant and chosen a priori. In this section, we will develop an approach that chooses the time step size Δt_n adaptively (and differently) in every time step.

A good choice of Δt_n has to strike a compromise between not being too large (to ensure good accuracy) and being too small (to avoid excessive

cost). We will control accuracy through the local truncation error, ignoring contributions to the error from previous time steps:

$$\frac{|u(t_{n+1}) - u^{n+1}|}{\Delta t}.$$

The obvious problem with this formula is that we do not know the exact solution $u(t_{n+1})$. To address, we will replace it by a proxy – an approximation computed with a more accurate method, that is, a method of higher order. For example, suppose that the forward Euler method is used to compute u^{n+1}. Then one could use the Heun method to compute

$$\hat{u}^{n+1} = u^n + \frac{\Delta t_n}{2} f(t_n, u^n) + \frac{\Delta t_n}{2} f(t_n + \Delta t_n, u^n + \Delta t_n f(t_n, u^n))$$

and replace $u(t_{n+1})$ by $\hat{u}^{n+1}$:

$$\hat{\tau}_n = \frac{|\hat{u}^{n+1} - u^{n+1}|}{\Delta t_n}.$$

Given the estimate $\hat{\tau}_n$, we accept the approximation produced by the time step size Δt_n if $\hat{\tau}_n \leq \mathsf{tol}/T$ for a prescribed tolerance $\mathsf{tol} > 0$. If, however, $\hat{\tau}_n > \mathsf{tol}/T$, we reduce Δt_n (for example, dividing it by two) and repeat the process until we have found an acceptable time step size.

It could also happen that $\hat{\tau}_n \ll \mathsf{tol}/T$, that is, the estimated error is a lot smaller than the tolerance. In this case, it is wise to increment the time step for the next time step (for example, by doubling it).

As a final twist, we expect that the Heun method will deliver better accuracy than the forward Euler method. Hence, we return the approximation by the Heun method (even if the error is controlled for the less accurate forward Euler method).

The described adaptive strategy is summarized in Algorithm 6.1.

6.6 Runge-Kutta methods

All the one-step methods studied in this chapter belong to the more general family of *Runge-Kutta methods*, which take the following form:

$$\begin{aligned} K_i &= f\Big(t_n + c_i \Delta t, u^n + \Delta t \sum_{j=1}^{s} a_{ij} K_j\Big), \qquad i = 1, \ldots, s \\ u^{n+1} &= u^n + \Delta t \sum_{i=1}^{s} b_i K_i, \quad n \geq 0. \end{aligned} \tag{6.19}$$

Such a Runge-Kutta method first computes the so called *stages* $K_1, \ldots, K_s$ and computes the next approximation u^{n+1} using a linear combination of these stages.

Algorithm 6.1: ODE solver with adaptive time stepping: Combination of forward Euler with Heun.

Data: $f(t,u)$, u_0, $[t_0,T]$, Δt_{init}, tol
Result: N, $[t_0,t_1,\ldots,t_N=T]$, $[\hat{u}_0,\hat{u}_1,\ldots,\hat{u}_N]$
$n=0$; $\Delta t_0=\Delta t_{\text{init}}$ `// initial time step;`
while $t_n<T$ **do**
  $u^{n+1}=u^n+\Delta t_n f(t_n,u^n)$ `// forward Euler;`
  $\hat{u}^{n+1}=u^n+\frac{\Delta t_n}{2}f(t_n,u^n)+\frac{\Delta t_n}{2}f(t_n+\Delta t_n,u^{n+1})$ `// Heun;`
  $\hat{\tau}_n=\frac{|\hat{u}^{n+1}-u^{n+1}|}{\Delta t_n}$ `// error estimate;`
  if $\hat{\tau}_n>\text{tol}/T$ **then** `// reduce time step size`
    $\Delta t_n=\Delta t_n/2$;
  else `// accept time step`
    $t_{n+1}=t_n+\Delta t_n$;
    if $\hat{\tau}_n<\text{tol}/2T$ **then** `// increase time step size`
      $\Delta t_{n+1}=\min\{2\Delta t_n,T-t_{n+1}\}$;
    else
      $\Delta t_{n+1}=\min\{\Delta t_n,T-t_{n+1}\}$;
    end
    $n=n+1$;
  end
end

A Runge-Kutta method is completely identified by the coefficients $\{a_{ij}\}$, $\{b_i\}$ and $\{c_i\}$ which are usually stored in a table (called Butcher table):

$$\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$$

The method is explicit if $a_{ij}=0$ for all $j\geq i$, as every stage K_i can be computed explicitly from the previous stages K_j, $j<i$. As mentioned above, such methods are only conditionally stable, at best. The Butcher tables for the methods discussed in this chapter are collected in Table 6.1 (check them as an exercise).

`Python` contains, within scipy.integrate, several implemented Runge-Kutta methods. For example, scipy.integrate.RK45 implements a variation of Algorithm 6.1. It uses a Runge-Kutta method of order 5 with $s=7$ stages to control the accuracy (and thus choose the time step size) for a nearly identical Runge-Kutta method of order 5.

0	0
	1

Forward Euler

1	1
	1

Backward Euler

0	0	
1	$\frac{1}{2}$	$\frac{1}{2}$
	$\frac{1}{2}$	$\frac{1}{2}$

Crank-Nicolson

0	0	
1	1	0
	$\frac{1}{2}$	$\frac{1}{2}$

Heun

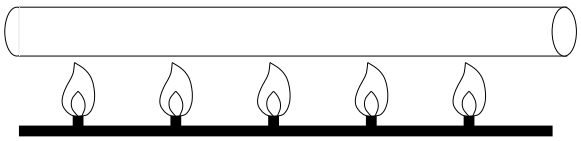
Table 6.1: Butcher tables corresponding to the forward Euler, backward Euler, Crank-Nicolson and Heun methods.

Chapter 7

Boundary value problems in one dimension

7.1 Example: Heat equation

Let us consider a metal bar of length L with density ρ and specific heat capacity c_p. We let $T(x,t)$ denote the temperature of the bar at the point $x \in [0,L]$ and at time t. Moreover, we let $J(x,t)$ denote the corresponding heat flow. Let us consider a heat source $f(x,t)$ generated, for example, by a flame:



The temperature variation in time in an infinitesimally small slice $[x, x+\Delta x]$ of the bar is given by

$$\rho c_p \frac{dT}{dt}(x,t)\Delta x = J(x,t) - J(x+\Delta x,t) + f(x,t)\Delta x.$$

Dividing by dx and taking the limit for $\Delta x \to 0$ results in

$$\rho c_p \frac{dT}{dt}(x,t) = -\frac{\partial J}{\partial x}(x,t) + f(x,t).$$

Fourier's law tells us that the heat flow is proportional to the temperature gradient:

$$J(x,t) = -k\frac{\partial T}{\partial x}(x,t), \tag{7.1}$$

with k being the heat conductivity. Combining both equations finally results in a differential model for describing the evolution of the temperature in the bar (*1D heat equation*):

$$\rho c_p \frac{dT}{dt}(x,t) - k\frac{\partial^2 T}{\partial x^2}(x,t) = f(x,t), \qquad x \in (0,L), \;\; t > 0. \tag{7.2}$$

This is a partial differential equation (PDE), with the unknown being a function, the temperature distribution $T(x,t)$ in the bar at any given time. To find a solution, we will have to provide the temperature distribution at the initial instant and specify what happens at both end points of the bar. For example, if the bar is in contact with heat tanks of constant temperature, we can add the *boundary conditions*

$$T(0,t) = T_l, \qquad T(L,t) = T_r, \qquad t > 0.$$

Boundary conditions of this type are called *Dirichlet boundary conditions*.

Another possibility is that the bar is thermally insulated. In this case, there is no heat flow at the end points of the bar and the boundary conditions are

$$J(0,t) = -k\frac{\partial T}{\partial x}(0,t) = 0, \qquad J(L,t) = -k\frac{\partial T}{\partial x}(L,t) = 0, \qquad t > 0.$$

These conditions are called *Neumann boundary conditions* and they depend on the value of the derivative of the solution at the end points, instead of the value of the solution itself.

Of course, we can also consider a mix of both types of conditions and consider, for example, a Dirichlet condition at the left end point and a Neumann condition at the right end point.

If we are only interested in the temperature distribution in equilibrium state (assuming that the heat source does not change over time), we can solve the associated *stationary problem*

$$-k\frac{\partial^2 T}{\partial x^2}(x) = f(x), \qquad x \in (0,L), \tag{7.3}$$

either with Dirichlet boundary conditions:

$$\begin{cases} -k\frac{\partial^2 T}{\partial x^2}(x) = f(x), & x \in (0,L), \\ T(0) = T_l, \;\; T(L) = T_r, \end{cases} \tag{7.4}$$

or with Neumann boundary conditions:

$$\begin{cases} -k\frac{\partial^2 T}{\partial x^2}(x) = f(x), & x \in (0,L), \\ k\frac{\partial T}{\partial x}(0) = J_l, \;\; k\frac{\partial T}{\partial x}(L) = J_r. \end{cases} \tag{7.5}$$

Equation (7.3) is a second order differential equation because it features the second derivative with respect to x. Problems of the form (7.4) and (7.5) are called *boundary value problems*.

7.2 Finite differences approximation of the stationary heat problem

We now consider the numerical solution of the one-dimensional stationary heat problem with Dirichlet boundary conditions. To simplify notation, we will denote the unknown function as $u(x)$, set $k = 1$, and denote the Dirichlet conditions by α and β:

$$\begin{cases} -\dfrac{\partial^2 u}{\partial x^2}(x) = f(x), & x \in (0, L), \\ u(0) = \alpha, \quad u(L) = \beta. \end{cases} \tag{7.6}$$

To find an approximated solution, we will proceed as follows: we divide the interval $[0, L]$ in $n+1$ sub-intervals $I_j = [x_{j-1}, x_j]$ of length $h = \frac{L}{n+1}$, where $x_j = jh$, $j = 1, \ldots, n$, and we seek for an approximation $u_j \approx u(x_j)$ at the nodes x_j.

At the end points $x_0 = 0$ and $x_{n+1} = L$, the solution is known because of the imposed Dirichlet boundary conditions. This means that as we set $u_0 = \alpha$ and $u_{n+1} = \beta$, it remains to determine the values u_j at the internal nodes x_j, with $j = 1, \ldots, n$.

At every internal node, the exact solution satisfies the equation

$$-\frac{\partial^2 u}{\partial x^2}(x_j) = f(x_j).$$

The idea is now to replace the exact second derivative by the finite difference approximation

$$\frac{\partial^2 u}{\partial x^2}(x_j) \approx \frac{u(x_{j-1}) - 2u(x_j) + u(x_{j+1})}{h^2}.$$

We then get the following scheme:

$$\begin{cases} \dfrac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), & j = 1, \ldots, n, \\ u_0 = \alpha, \quad u_{n+1} = \beta. \end{cases} \tag{7.7}$$

Taking into account the boundary conditions, the first and last equation can be re-written as follows

$$\begin{aligned} \frac{-u_0 + 2u_1 - u_2}{h^2} = f(x_1) \quad &\Longleftrightarrow \quad \frac{2u_1 - u_2}{h^2} = f(x_1) + \frac{\alpha}{h^2}, \\ \frac{-u_{n-1} + 2u_n - u_{n+1}}{h^2} = f(x_n) \quad &\Longleftrightarrow \quad \frac{-u_{n-1} + 2u_n}{h^2} = f(x_n) + \frac{\beta}{h^2}, \end{aligned}$$

and, in turn, the system (7.7) becomes

$$\begin{cases} \dfrac{2u_1 - u_2}{h^2} = f(x_1) + \dfrac{\alpha}{h^2} & \text{for } j = 1, \\ \dfrac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), & \text{for } j = 2, \ldots, n-1, \\ \dfrac{-u_{n-1} + 2u_n}{h^2} = f(x_n) + \dfrac{\beta}{h^2} & \text{for } j = n. \end{cases} \tag{7.8}$$

If we introduce the unknowns vector $\mathbf{u} = [u_1, \ldots, u_n]^\top$, the linear system (7.8) can be written in matrix form

$$A\mathbf{u} = \tilde{\mathbf{f}},$$

with matrix $A \in \mathbb{R}^{n\times n}$ and the right-hand side $\tilde{\mathbf{f}} \in \mathbb{R}^n$ given by

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2 \end{bmatrix}, \qquad \tilde{\mathbf{f}} = \begin{bmatrix} f(x_1) + \frac{\alpha}{h^2} \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \\ f(x_n) + \frac{\beta}{h^2} \end{bmatrix}. \tag{7.9}$$

Therefore, in order to solve Problem (7.6) approximately using finite differences, we have to solve a linear system of equations. We notice that the matrix A is symmetric and tridiagonal. Moreover, we can prove that it is also positive definite. This means we can use the LU factorization method (or Cholesky factorization) in a very efficient way.

Additionally, one can prove that the condition number of the matrix A satisfies

$$\kappa(A) = O(h^{-2}),$$

that is, the matrix A becomes increasingly ill-conditioned as h becomes smaller (when increasing the number of discretization points).

7.2.1 Stability and error analysis

For System (7.8), one can prove the following stability result:

Theorem 7.1. *For all* $\mathbf{f} = [f(x_1), \ldots, f(x_n)]^\top \in \mathbb{R}^n$ *and every* $\alpha, \beta \in \mathbb{R}$, *the system* (7.8) *admits a unique solution* $\mathbf{u} = [u_1, \ldots, u_n]^\top \in \mathbb{R}^n$ *which satisfies*

$$\max_{j=1,\ldots,n} |u_j| \leq \frac{1}{8} \max_{j=1,\ldots,n} |f(x_j)| + \max\{|\alpha|, |\beta|\}. \tag{7.10}$$

This result tells us that the norm of the solution $\mathbf{u}$ cannot become too big; it is controlled by the norm of the vector $\mathbf{f}$ and the values α, β at the boundaries.

We are now interested in studying the behavior of the error $e_j = u(x_j) - u_j$, $j = 1, \ldots, n$, associated to the finite differences approximation (7.8) in terms of h (or, equivalently, in terms of the number of discretization points). For this purpose, we introduce notions analogous to the ones already used in the study of numerical schemes for ODEs (see Chapter 6).

Definition 7.1. *A numerical scheme approximating the boundary value problem* (7.6) *is* ***convergent of order*** *p if there exists a constant* $C > 0$ *such that*

$$\max_{j=0,\dots,n+1} |u(x_j) - u_j| \leq Ch^p,$$

provided that the solution is sufficiently smooth.

We also introduce a corresponding notion of local (in space) truncation error.

Definition 7.2 (**Local truncation error**)**.** *Consider the exact solution* $u(x_j)$ *of Problem* (7.6) *at the nodes* $x_j = jh$ *for* $j = 0, \dots, n+1$*. Then the* local truncation error *at the node* x_j *of the finite differences scheme* (7.8) *is defined as*

$$\tau_j = \frac{-u(x_{j-1}) + 2u(x_j) - u(x_{j+1})}{h^2} - f(x_j), \quad j = 1, \dots, n. \tag{7.11}$$

As the exact solution satisfies $-\frac{\partial^2 u}{\partial x^2}(x_j) = f(x_j)$, the local truncation error

$$\tau_j = -\frac{u(x_{j-1}) - 2u(x_j) + u(x_{j+1})}{h^2} + \frac{\partial^2 u}{\partial x^2}(x_j)$$

represents the approximation of the second derivative of u using finite differences. We have the following result (the proof is left as an exercise)

Result 7.1. *The local truncation error* (7.11) *satisfies*

$$\max_{j=1,\dots,n} |\tau_j| \leq \frac{h^2}{12} \max_{x\in[0,L]} |u''''(x)|, \tag{7.12}$$

provided that the exact solution is four times continuously differentiable.

According to the definition of local truncation error, the exact solution satisfies the discrete problem

$$\frac{-u(x_{j-1}) + 2u(x_j) - u(x_{j+1})}{h^2} = f(x_j) + \tau_j, \qquad j = 1, \dots, n, \tag{7.13}$$

while the approximated solution u_j satisfies the problem

$$\frac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), \qquad j = 1, \dots, n. \tag{7.14}$$

If we subtract (7.14) from (7.13), we get the following system of equations for the error:

$$\begin{cases} \frac{-e_{j-1}+2e_j-e_{j+1}}{h^2} = \tau_j, \qquad j = 1, \dots, n, \\ e_0 = e_{n+1} = 0. \end{cases} \tag{7.15}$$

This system has the same form of (7.7), or equivalently, of System (7.8), with the local truncation error on the right-hand-side and with homogeneous boundary conditions ($\alpha = \beta = 0$). This allows us to apply the stability result of Theorem 7.1 to conclude

$$\max_{j=1,\dots,n} |e_j| \leq \frac{1}{8} \max_{i=1,\dots,n} |\tau_i|,$$

which gives an estimation of the error of the solution in terms of the maximal local truncation error.

The following theorem summarizes the results obtained.

Theorem 7.2. *The finite differences scheme* (7.8) *applied to* (7.6) *converges with second order. More specifically, we have*

$$\max_{j=1,\dots,n} |u(x_j) - u_j| \leq Ch^2, \tag{7.16}$$

where $C = \frac{1}{96} \max_{x\in[0,L]} |u''''(x)|$*, provided that the exact solution is four times continuously differentiable.*

Thanks to the stability result of Theorem 7.1, we notice that the error $\max_{j=1,\dots,n} |u(x_j) - u_j|$ has the same order, in h, as the local truncation error. This conclusion is also true for several other boundary value problems and finite differences discretizations. Indeed, it often suffice to study the local truncation error to predict the order of the method.

7.2.2 Neumann boundary conditions

Let us now consider a case of mixed Dirichlet–Neumann boundary conditions:

$$\begin{cases} -\dfrac{\partial^2 u}{\partial x^2}(x) = f(x), & x \in (0, L), \\ u(0) = \alpha, \quad u'(L) = \beta. \end{cases} \tag{7.17}$$

As in the previous section, we divide the interval $[0, L]$ in $n+1$ sub-intervals $I_j = [x_{j-1}, x_j]$ of length $h = \frac{L}{n+1}$, where $x_j = jh$, $j = 0, \dots, n+1$, and we let $u_j \approx u(x_j)$ denote the approximate solution at node x_j.

This time, the solution at the node x_{n+1} (right extremity) is not known, which means that the vector of unknowns is $\mathbf{u} = [u_1, \dots, u_{n+1}]^\top \in \mathbb{R}^{n+1}$.

At every internal node x_j, $j = 1, \dots, n$, we write the approximated equation

$$\frac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), \qquad j = 1, \dots, n.$$

We still have to determine the equation for the right end point (at the node x_{n+1}) and how to discretize the Neumann boundary condition $u'(L) = \beta$. To this end, we have two possibilities which are presented below.

First method: First order finite difference

The first idea is to discretize the Neumann condition $u'(x_{n+1}) = \beta$ using a finite difference. As the vector of unknowns contains the values $u_1, \ldots, u_{n+1}$, it makes sense to backward finite differences, yielding the equation

$$\frac{u_{n+1} - u_n}{h} = \beta. \tag{7.18}$$

Note that this formula is only of first order! Using this choice, we arrive at the following system of $n+1$ equations:

$$\begin{cases} \dfrac{2u_1 - u_2}{h^2} = f(x_1) + \dfrac{\alpha}{h^2}, & \text{for } j = 1, \\ \dfrac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), & \text{for } j = 2, \ldots, n, \\ \dfrac{-u_n + u_{n+1}}{h} = \beta, & \text{for } j = n+1, \end{cases}$$

which can be written in matrix form

$$A\mathbf{u} = \tilde{\mathbf{f}},$$

with

$$A = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 1 \end{bmatrix}, \qquad \tilde{\mathbf{f}} = \begin{bmatrix} f(x_1) + \frac{\alpha}{h^2} \\ f(x_2) \\ \vdots \\ f(x_n) \\ \frac{\beta}{h} \end{bmatrix}. \tag{7.19}$$

As mentioned, the drawback of this approach is that the approximation (7.18) is only of first order, that is, the local truncation error at the node x_{n+1} satisfies

$$|\tau_{n+1}| = \left| \frac{u(x_{n+1}) - u(x_n)}{h} - \beta \right| = \left| u'(L) + \frac{h}{2}u''(\xi) - \beta \right| \leq \frac{h}{2} \max_{x \in [x_n, x_{n+1}]} |u''(x)|.$$

In turn, the approximation of Problem (7.17) is also only of first order.

Second method: Ghost node

To retrieve a second order approximation, we would like to use a centered finite difference in x_{n+1}. For this purpose, we introduce the node $x_{n+2} = (n+2)h$, which is outside the interval $[0, L]$, and the corresponding value u_{n+2}, so that we can write the centered finite difference

$$\frac{u_{n+2} - u_n}{2h} = \beta, \quad \text{for } j = n+1. \tag{7.20}$$

Because we added the new unknown u_{n+2}, we also have to add an equation. At node x_{n+1}, the finite differences discretization of the equation $-u'' = f$ can be written as:

$$\frac{-u_n + 2u_{n+1} - u_{n+2}}{h^2} = f(x_{n+1}). \tag{7.21}$$

From Equation (7.20), we get $u_{n+2} = u_n + 2h\beta$ which we insert in (7.21) to obtain

$$\frac{-2u_n + 2u_{n+1}}{h^2} = f(x_{n+1}) + \frac{2\beta}{h}. \tag{7.22}$$

We finally arrive at the following system of $n+1$ equations:

$$\begin{cases} \dfrac{2u_1 - u_2}{h^2} = f(x_1) + \dfrac{\alpha}{h^2}, & \text{for } j = 1, \\ \dfrac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f(x_j), & \text{for } j = 2, \ldots, n, \\ \dfrac{-u_n + u_{n+1}}{h^2} = \dfrac{1}{2} f(x_{n+1}) + \dfrac{\beta}{h}, & \text{for } j = n+1, \end{cases} \tag{7.23}$$

which can be written in matrix form as $A\mathbf{u} = \tilde{\mathbf{f}}$ with

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 1 \end{bmatrix}, \qquad \tilde{\mathbf{f}} = \begin{bmatrix} f(x_1) + \frac{\alpha}{h^2} \\ f(x_2) \\ \vdots \\ f(x_n) \\ \frac{1}{2} f(x_{n+1}) + \frac{\beta}{h} \end{bmatrix}. \tag{7.24}$$

We notice that the node x_{n+2} and the corresponding unknown u_{n+2} have both been introduced only to be able to write the centered finite difference (7.20), but they do not appear in the final system (7.23). The node x_{n+2} is called *ghost node*.

Notice also that the only difference between System (7.24) and System (7.19) can be found in the last component of the vector $\mathbf{f}$, the matrix A being the same in both cases. However, this small difference is sufficient to produce a method of second order!