MAS235

Introduction to Numerical Computing

Hugo Touchette

School of Mathematical Sciences,
Queen Mary, University of London

Version of October 17, 2008, 14:3

Course description

This course investigates the use of computer algebra, numerical techniques and computer graphics as tools for developing the understanding and the solution of a number of problems in the mathematical sciences. Topics that will be addressed will include: linear algebra, the solution of algebraic equations, the generation and use of quadrature rules and the numeric solution of differential equations and, time permitting, some other aspects of computational mathematics. The computer language used for the course is Maple.

Course content

- Revision of Maple [Week 1]
- Chapter 1: Numerical approximations and errors [Week 2]
- Chapter 2: Numerical solutions of algebraic equations [Weeks 3-5]
- Chapter 3: Calculation of eigenvalues and eigenvectors [Weeks 6-8]
- Chapter 4: Numerical integration [Weeks 9-10]
- *Chapter 5: Numerical solutions of differential equations [Week 11]
- Programming languages, revision lecture, practice exam [Week 12]

Optional textbooks

For revising Maple:

- F. J. Wright, Computing with Maple, Chapman/CRC Press, 2001.
- F. Vivaldi, Experimental Mathematics with Maple, CRC Press, 2001.

Specific to the content of this course:

- R.L. Burden, J.D. Faires, Numerical Analysis, Prindle, Weber & Schmidt, 1985. Available at the main library: QA297 BUR

Sources

These notes are based on lecture notes prepared by Francis Wright for the course Computational Mathematics III (MAS225), as well as previous lecture notes written by Andrew Tworkowski for the course Experimental Mathematics.

Contents

1 Numerical approximations and errors **5**
1.1 Rounding errors . . . 5
1.2 Absolute and relative errors . . . 6
1.3 Error propagation . . . 6
1.4 Cancellation errors . . . 7
1.5 Non-associativity of floating-point addition . . . 8
1.6 Truncation errors . . . 8
1.7 Error combination . . . 9
1.8 Stable and unstable iterations . . . 9
1.9 Ill-conditioned and ill-posed problems . . . 10

2 Numerical solutions of algebraic equations **13**
2.1 Introduction . . . 13
2.2 Fixed-point iteration method . . . 13
Convergence of the fixed-point iteration method . . . 14
Summary of the method . . . 15
2.3 *Bisection method . . . 16
2.4 Newton-Raphson method . . . 19
Summary of the method . . . 20
2.5 Variants of the Newton-Raphson method . . . 22
Numerical estimation of the derivative . . . 22
Root suppression . . . 22
Closely spaced roots . . . 23
Complex roots . . . 23
2.6 Iteration method for systems of many equations . . . 24
Summary of vector notation . . . 24
Fixed-point method in many-dimensions . . . 25
Convergence of the fixed-point iteration method . . . 25
2.7 Newton-Raphson method in many dimensions . . . 26

2.8 *Stability condition using the spectral norm 28

3 Numerical computation of eigenvalues and eigenvectors **29**
3.1 Short revision of matrix algebra 29
3.2 Power method 31
Stopping rule 33
Normalisation step 34
Choice of the initial vector 34
3.3 Deflation method 34
Eigenvectors of A and A_p 40
Deflation of the matrix A_p 40
Recursive call of Wielandt's deflation 41
3.4 $\boldsymbol{QR}$ algorithm 42

4 Numerical integration **49**
4.1 Introduction 49
4.2 Riemann sums 50
4.3 Trapeze integration rule 52
Construction of the trapeze rule 54
Composite trapeze rule 55
Estimate of the error 55
4.4 Simpson's integration rule 57
Construction of Simpson's rule 58
Composite Simpson's rule 59
Estimation of the error 59
4.5 *Newton-Cotes integration 61
Lagrange interpolating polynomials 62
n-point Newton-Cotes integration 63
4.6 *Adaptive integration 65

Chapter 1

Numerical approximations and errors

1.1. Rounding errors

Integers such as 1, -45 or 2^{10} can be represented exactly on a computer, which is to say that their representation on a computer does not involve any approximation.

Rational numbers such as $\frac{1}{2}$ or $-\frac{3}{415}$ can also be represented exactly if they are kept internally (in Maple's memory or "kernel") as fractions.

Irrational numbers such as $\pi = 3.14159...$, $\sqrt{2} = 1.4142\ldots$ or 0.333... cannot be represented exactly on computers. They can only be **approximated** with **truncated decimal representations**. For example, 0.333... (0 followed by an infinite number of 3's) can be represented with 10 digits as 0.333333333 (0 followed by nine 3's). The loss of accuracy in representing an irrational number by some truncated (rational) number is called the **rounding error**.

The floating-point representation of a number has the form

$$x = \text{ mantissa } \times \text{ base}^{\text{exponent}}. \tag{1.1}$$

Usually the base is 10.

▶ Examples in class.

Remarks about Maple:

- Numbers such as π or $\sqrt{2}$ can be handled exactly (without rounding errors) by Maple at a symbolic level. The Maple (protected) symbol for π is `Pi`.

- The number of digits used by Maple for representing floating-point numbers is set by the global and protected variable `Digits`. Writing

```
>Digits:=20;
```

  changes the number of digits used by Maple to 20. The default value is 10.

- `1/3` and `1.0/3.0` are two different expressions in Maple. The first is the exact fraction, the second is the floating-point evaluation of the first, for example, `.3333333333` with `Digits:=10`). An equivalent syntax for `1.0/3.0` is `1./3.`

- `evalf[n](x)` outputs the floating-point representation of x using n digits. n is an optional argument; if it is not used, Maple uses the current number of digits set by `Digits`. For example,

```
>evalf(sqrt(2));
```

  yields `1.414213562` in the default precision of 10 digits.

1.2. Absolute and relative errors

Given a number x and an **approximation** x^* of x, we define the **absolute error** δx between x and x^* as the magnitude of the difference between x and x^*:

$$\delta x = |x - x^*|. \tag{1.2}$$

The **relative error** r_x between x and its approximation x^* is defined as the error relative to the magnitude of x, that is,

$$r_x = \frac{|x - x^*|}{|x|}. \tag{1.3}$$

▶ Examples in class.

1.3. Error propagation

Suppose that we want to calculate a given function $f(x)$, but that instead of calculating f exactly at x, we calculate this function at an approximation x^* of x. The absolute error δf associated with this calculation is, by definition,

$$\delta f = |f(x) - f(x^*)|. \tag{1.4}$$

How is this error related to the error that we have on x, that is, the absolute error $\delta x = |x - x^*|$?

If the function $f(x)$ is exactly entered in Maple and is differentiable, then we can make the following approximation:

$$\delta f \approx |f'(x)|\delta x, \tag{1.5}$$

where $f'(x)$ is the derivative of f at x.

This approximation is derived using the Taylor series of $f(x)$:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \cdots. \tag{1.6}$$

Using $x_0 = x^* = x + \delta x$, and truncating the Taylor series to first order in δx, we write

$$\begin{aligned} \delta f &= |f(x) - f(x + \delta x)| \\ &\approx |f(x) - f(x) - f'(x)\delta x| \qquad \text{(valid to first order in } \delta x) \\ &= |f'(x)|\delta x. \end{aligned} \tag{1.7}$$

The relative error is calculated similarly, to first order in δx, as

$$r_f = \left|\frac{\delta f}{f}\right| = \left|\frac{f'(x)}{f}\right| \delta x = \left|\frac{x}{f}\right| |f'(x)| \left|\frac{\delta x}{x}\right| = \left|\frac{x}{f}\right| |f'(x)| r_x. \tag{1.8}$$

Similar estimates exist for functions of more than one variables. For $f(x, y)$, for example, we have

$$\delta f \approx \left|\frac{\partial f}{\partial x}\right| \delta x + \left|\frac{\partial f}{\partial y}\right| \delta y \tag{1.9}$$

to first order in δx and δy.

▶ Examples in class.

1.4. Cancellation errors

Consider $f(x, y) = x - y$. Using Eqn. (1.9), we obtain

$$r_{x-y} = \left|\frac{x}{x-y}\right| r_x + \left|\frac{y}{x-y}\right| r_y. \tag{1.10}$$

This formula shows that the relative error is likely to be large when substracting two similar values, that is, two values x and y such that $x - y$ is close to 0. This occurs because the accurate digits of x and y cancel out, leaving a tiny number which may not be representable using the accuracy set in Maple.

Example 1.4.1. `12345.678912-12345.678911` yields `0.` with `Digits:=10`. This obviously wrong result is explained by noting that the answer requires 11 digits to be correctly represented:

$$12345.678912 - 12345.678911 = 00000.000001. \tag{1.11}$$

Another type of cancellation error occurs when adding very small numbers with very big numbers.

Example 1.4.2. `1e-5+1e5` yields the wrong number `100000.0000` with `Digits:=10`. The correct answer requires 11 digits.

1.5. Non-associativity of floating-point addition

Cancellation errors imply that the addition, when done numerically, may be **non-associative**, which is to say that

$$(x+y)+z, \qquad (x+y \text{ followed by } +z) \tag{1.12}$$

may not be equal to

$$x+(y+z), \qquad (y+z \text{ followed by } +x) \tag{1.13}$$

Example 1.5.1. Take `x = 1e-5`, `y = 1e5` and `z = 1e-5 - 1e5` and convince yourself that $(x+y)+z \neq x+(y+z)$ with Maple's default accuracy. There is a cancellation error in doing $x+y$ (see the previous example) which carries over when adding z. Calculating $y+z$ and then adding x carries no error.

The general rule for minimizing errors when doing additions (and substractions) is to perform addition (and substraction) on values of order of decreasing magnitude: start with the big numbers to finish with the small ones.

1.6. Truncation errors

Truncation errors arise when truncating infinite sums or products (or any infinite expressions in general).

Example 1.6.1. The exponential function e^x can be represented by an infinite Taylor series:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}. \tag{1.14}$$

This representation of e^x cannot be represented on a computer. In order to use it, we must truncate it, for example, to order n:

$$T_n(x) = \sum_{r=0}^{n} \frac{x^r}{r!} \tag{1.15}$$

We call T_n an n**th-order truncation** of e^x.

The **truncation error** is the bit that is truncated in a truncation. Thus, if T_n is a truncation of some expression T, then the absolute truncation error is

$$\delta_n = |T_n - T|. \tag{1.16}$$

We put the subscript n to emphasize that the truncation error depends on the truncation order.

In most cases of interest we cannot calculate the error δ_n because we have access only to the approximation T_n and not its exact value T. In this case, we can approximate the truncation error δ_n itself by

$$\delta_n \approx |T_n - T_{n-1}|, \tag{1.17}$$

where T_n is the truncation of T to order n, while T_{n-1} is the truncation of T to order $n-1$. To justify this formula, note that the difference δ_n is the added accuracy that you obtain when considering T_n rather than T_{n-1} as your approximation of T (assuming that T_n converges to T as $n \to \infty$). Seen from another point of view, δ_n is the error that results from considering T_{n-1} rather than T_n as your approximation of T.

1.7. Error combination

The **total error** of a computation is the combination of the rounding errors and truncation errors.

▶ Examples in class.

1.8. Stable and unstable iterations

Example 1.8.1. We wish to numerically evaluate

$$I_n = \int_0^1 \frac{x^n}{x+10} dx. \tag{1.18}$$

With a bit of ingenuity, we can arrive at the following recurrence relation for I_n:

$$I_n + 10 I_{n-1} = \frac{1}{n}. \tag{1.19}$$

This suggests computing I_n in an iterative fashion:

$$I_0 \to I_1 = 1 - 10I_0 \to I_2 = \frac{1}{2} - 10I_1 \to \cdots \to I_n = \frac{1}{n} - 10I_{n-1}, \quad (1.20)$$

starting with the known value of I_0:

$$I_0 = \int_0^1 \frac{1}{x+10}\, dx = \ln(x+10)|_0^1 = \ln 11 - \ln 10 = \ln(1.1). \quad (1.21)$$

Implementing this iterative calculation of I_n in Maple using 10 digits precision, we arrive, after 10 iterates, at $I_{10} = -0.0345870110$. This result is obviously wrong because $I_n > 0$ for all $n \geq 0$. But what goes wrong exactly? The problem is that, by calculating I_n using its previous value I_{n-1}, we magnify the error on the calculation of I_{n-1} by a factor 10 because of the 10 appearing in the recurrence relation:

$$I_n = \frac{1}{n} - 10I_{n-1}. \quad (1.22)$$

After n iterations, the error on I_0 is then magnified by a factor 10^n. Thus, if the original error δ_0 on I_0 is 10^{-10}, then after 10 iterations, we obtain $\delta_{10} = \delta_0 \times 10^n = 10^{-10} \times 10^{10} = 1$.

The iterative calculation of the previous example is said to be an **unstable iteration** because the error grows at each step of the iteration. A **stable iteration** is an iteration in which the error is not magnified at each step, that is, in which the error decreases or stays the same at each step.

Example 1.8.2. The backward iteration

$$I_{n-1} = \frac{1}{10}\left(\frac{1}{n} - I_n\right) \quad (1.23)$$

is stable. At each step, the error is decreased by a factor 10.

1.9. Ill-conditioned and ill-posed problems

An **ill-conditioned problem** is a problem for which a small change in the input results in a large (and often uncontrollable) change in the output.

Example 1.9.1. The inverse of the matrix

$$A = \begin{pmatrix} 1 & 0 \\ 0 & \varepsilon \end{pmatrix} \quad (1.24)$$

is

$$A^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & \varepsilon^{-1} \end{pmatrix}. \tag{1.25}$$

The calculation of A^{-1} is ill-conditioned because a small change in ε (for example, $\varepsilon = 10^{-4}$ compared to $\varepsilon = 10^{-5}$) results in a large change in the calculation of A^{-1} ($\varepsilon^{-1} = 10^4$ compared to $\varepsilon^{-1} = 10^5$). If we do not set the accuracy right, there may be some cancellation errors.

An **ill-posed** or **ill-defined problem** is a problem having no meaningful numerical solution.

Example 1.9.2. The Maple function `signum(x)` outputs the sign of x with the result 0 if $x = 0$. At the symbolic level, this function is well defined. For example,

```
>signum(sin(Pi));
```

yields 0 as expected. The same function, however, is ill-defined when using floating-point numbers. For example,

```
>signum(evalf(sin(Pi)));
```

yields not 0 but -1. The wrong answer arises because the floating-point evaluation of $\sin(\pi)$ is not exactly zero.

Chapter 2

Numerical solutions of algebraic equations

2.1. Introduction

An **algebraic equation** is an equation of the form $f(x) = 0$ involving any of the known **algebraic functions**: $\sin x$, $\cos x$, e^x, x^n, $\sqrt{x}$, etc.

Given a smooth algebraic function $f : \mathbb{R} \to \mathbb{R}$, we seek to solve the equation

$$f(x) = 0. \tag{2.1}$$

A solution of this equation is called a **zero** or a **root**, and will be denoted by x^*. There may be more than one root; there may be none. A root x^* is obviously such that $f(x^*) = 0$.

Remark 2.1.1. The symbolic equation solver of Maple is `solve`. To force a numerical evaluation of `solve`, use `fsolve`. The syntax of both functions is well described in the help section of Maple.

2.2. Fixed-point iteration method

The **fixed-point iteration method** consists in choosing an **iteration function** $\Phi(x)$, also called a **map**, from which we build successive **iterates** $x_1, x_2, ..., x_n$ following the equation

$$x_n = \Phi(x_{n-1}). \tag{2.2}$$

The iteration starts with a 0th value x_0 called the **initial value** or **seed**. A chain of **iterates** $x_1, x_2, ..., x_n$ is thus generated in the following way:

$$x_0 \to x_1 = \Phi(x_0) \to x_2 = \Phi(x_1) \to \cdots \to x_n = \Phi(x_{n-1}). \tag{2.3}$$

The idea of the iteration method is to choose $\Phi(x)$ such that x_n approaches x^* as $n \to \infty$. In terms of the absolute error $\delta_n = |x_n - x^*|$, this means that we are seeking an iteration function $\Phi(x)$ such that

$$\delta_n = |x_n - x^*| \to 0 \text{ as } n \to \infty. \tag{2.4}$$

Equivalently, we need to choose $\Phi(x)$ so that $\delta_n < \delta_{n-1}$.

Convergence of the fixed-point iteration method

There are two conditions ensuring that the chain of iterates $\{x_i\}_{i=1}^n$ built from $\Phi(x)$ converges to the desired root x^*:

Condition 1: (Fixed-point condition). The root x^* must be a **fixed-point** of $\Phi(x)$, by which we mean that $\Phi(x^*) = x^*$.

This condition arises because the iteration process must stay at x^* if it reaches x^* at a certain point, that is, if $x_n = x^*$ for some $n \geq 0$. In particular, the iteration must stay at x^* if started at x^*, that is, if $x_0 = x^*$.

Given $f(x) = 0$, an obvious choice of map $\Phi(x)$ having the root(s) of $f(x)$ as its fixed-point(s) is $\Phi(x) = f(x) + x$. Indeed,

$$\Phi(x^*) = \underbrace{f(x^*)}_{=0} + x^* = x^*. \tag{2.5}$$

Condition 2: (Stability condition). $|\Phi'(x^*)| < 1$.

This conditions ensures that the error δ_n decreases as $n \to \infty$.

Proof of Condition 2: The correct choice of $\Phi(x)$ is such that $\delta_{n+1} < \delta_n$, that is,

$$|\Phi(x_n) - x^*| < |x_n - x^*|. \tag{2.6}$$

This is equivalent to

$$|\Phi(x_n) - \Phi(x^*)| < |x_n - x^*|. \tag{2.7}$$

since $\Phi(x^*) = x^*$ (x^* is a fixed-point of $\Phi(x)$). Now, assuming that x_n is near x^*, we can develop $\Phi(x_n)$ in a Taylor series around x^*:

$$\Phi(x_n) = \Phi(x^*) + \Phi'(x^*)(x_n - x^*) + \frac{\Phi''(x^*)}{2}(x_n - x^*)^2 + \cdots. \tag{2.8}$$

Next, as we did when calculating the error δf, we truncate the series to the first-order term in $(x_n - x^*)$ to obtain the approximation

$$\Phi(x_n) \approx \Phi(x^*) + \Phi'(x^*)(x_n - x^*). \tag{2.9}$$

The absolute error δ_{n+1} for the $(n+1)$th iterate is thus approximated as

$$|\Phi(x_n) - \Phi(x^*)| \approx |\Phi'(x^*)||x_n - x^*|, \tag{2.10}$$

that is,

$$\delta_{n+1} \approx |\Phi'(x^*)|\delta_n. \tag{2.11}$$

This shows that if we want the error to decrease as n increases, we must require that $|\Phi'(x^*)| < 1$.

Summary of the method

If we can find a map $\Phi(x)$ satisfying Conditions 1 and 2, then

$$x_n = \Phi(x_{n-1}) \to x^* \tag{2.12}$$

as $n \to \infty$ starting from any initial value x_0 chosen in a neighborhood of x^*.

In practice, we have to stop generating iterates when we get close enough to x^* or when we think that we are close enough to x^*. A possible **stopping rule** is to choose a large enough value of n such that $x_n \approx x^*$, but since x^* is generally unknown, a better stopping rule consists in stopping when the iteration error δ_n, estimated as $\delta_n \approx |x_n - x_{n-1}|$, becomes smaller than some **threshold** value ε. This means that we generate iterates until $|x_n - x_{n-1}| < \varepsilon$. The last generated iterate is taken as the approximation of x^*; the error on that value is the **truncation error**, that is, ε.

Example 2.2.1. Suppose we want to solve the equation

$$f(x) = \cos x - x = 0. \tag{2.13}$$

We transform this equation to

$$\cos x = x. \tag{2.14}$$

It is easy to see graphically that the two functions $\cos x$ and x intersect at some point $x^* \in (0, \pi/2)$ and only at that point. Therefore, the function $f(x)$ has only one root located somewhere between $x = 0$ and $x = \pi/2$. To estimate the root, we can use $\Phi(x) = \cos x$ as our iteration function. This

choice of map obviously verifies Condition 1, since $\Phi(x^*) = \cos(x^*) = x^*$ according to Eqn. (2.14). It also satisfies Condition 2 because

$$|\Phi'(x)| = |(\cos x)'| < 1 \text{ for } x \in (0, \pi/2). \tag{2.15}$$

With these conditions verified, we know that the chain of iterates built from $\Phi(x)$ will converge to x^*, provided that we choose x_0 sufficiently close to x^*. In the present case, the value of x_0 is actually unimportant: any initial seed leads to a chain of iterates that converge to x^*.

Maple code 2.2.1: Fixed-point iteration method

```
FPsolve:=proc(Phi::procedure,x0::realcons,tol::realcons)
   local i,x,xp,abserr,imax;
   imax:=10^6;
   x:=x0;
   abserr:=abs(x-Phi(x));
   for i while abserr>tol do
       xp:=x;
       x:=Phi(x);
       abserr:=abs(x-xp);
       if i>imax then
          error "Too many iterations!"
       end if;
    end do;
    x;
end proc:
```

2.3. *Bisection method

Consider a function $f(x)$ that intersects the x-axis once between the interval $[a, b]$; see Figure 2.1. The intersection is the root x^* of $f(x) = 0$.

The bisection method finds the root x^* by *squeezing* or *sandwiching* it in an interval $[a_n, b_n]$ which it is our goal to make smaller and smaller in an iterative way. This is done using the following steps:

Step 1: (Initialization). Set $i = 1$, $a_i = a$ and $b_i = b$. We know that x^* is in the initial interval $[a_1, b_1]$.

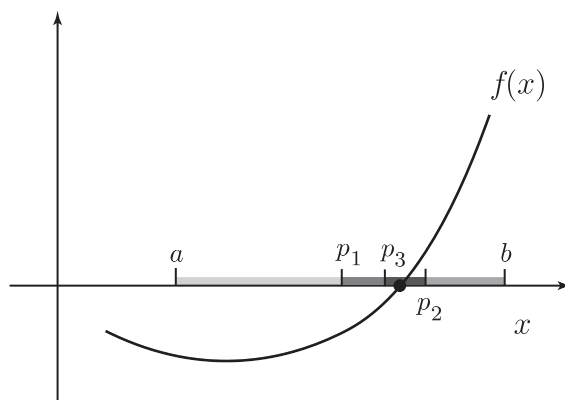

Figure 2.1: Illustration of the bisection method.

Step 2: (Begin squeezing). Calculate the middle point between a_i and b_i:

$$p_i = \frac{1}{2}(a_i + b_i). \tag{2.16}$$

Step 3: (Decision step). Decide whether the root x^* is to the left or to the right of the middle point p_i (it can also lie on p_i) and update the interval $[a_i, b_i]$ accordingly:

1. If $f(p_i) = 0$ then $x^* = p_i$; the root is found.
2. If $f(p_i)$ and $f(a_i)$ have the same sign, then x^* must be to the right of p_i, that is, in the interval $[p_i, b_i]$. Accordingly, update the interval by setting $a_{i+1} = p_i$ and $b_{i+1} = b_i$. The new interval to work with now is $[a_{i+1}, b_{i+1}] = [p_i, b_i]$.
3. If $f(p_i)$ and $f(b_i)$ have the same sign, then x^* must be to the left of p_i, that is, in $[a_i, p_i]$. Update the new interval to $[a_{i+1}, b_{i+1}] = [a_i, p_i]$.

Step 4: (Iteration). $i := i + 1$ and repeat Steps 2 and 3.

Step 5: (Stopping rule). Stop the iteration when the interval $[a_n, b_n]$ is small enough, for example, when $|a_i - b_i| < \varepsilon$. Another condition for stopping could be $|f(p_N)| < \varepsilon$ or $i = N$, i.e, stop after some pre-determined number N of steps. We could also stop when $f(p_i) = 0$; however, it is very unlikely that we can reach the root exactly in a finite number of steps.

The Maple code implementing these steps is shown next.

Maple code 2.3.1: Bisection method

```
BSsolve:=proc(f::procedure,a::realcons,b::realcons,tol::realcons)
   local i,imax,midpoint,intlength;
   imax:=10^6;
   intlength:=abs(a-b);
   for i while intlength>tol do
      midpoint:=(a+b)/2;
      if f(midpoint)=0 then
         return midpoint;
      elif f(midpoint)*f(a)>0 then
         a:=midpoint;
      else
         b:=midpoint;
      end if;
      intlength:=abs(a-b);
      if i>imax then
         error "Too many iterations!"
      end if
   end do;
   midpoint;
end proc:
```

2.4. Newton-Raphson method

The **Newton-Raphson method** is based on the following choice of iteration function:

$$x_n = \Phi(x_{n-1}) = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \tag{2.17}$$

where $f(x)$ is, as always, the function whose root(s) we are seeking. (Do not confuse at this point f with Φ.)

This choice of $\Phi(x)$ is justified as follows; see Figure 2.2. Choose an initial approximation x_0 of the root x^*. Assuming that x_0 is close enough to x^*, we can develop $f(x)$ to first order around x_0 to obtain the following approximation:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0). \tag{2.18}$$

Evaluating this expression at the fixed-point $x = x^*$, we obtain

$$f(x^*) = 0 \approx f(x_0) + f'(x_0)(x^* - x_0). \tag{2.19}$$

Then solving for x^* gives

$$x^* \approx x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{2.20}$$

The expression on the right-hand side of this equation is an approximation of x^* which should be a better approximation to x^* than x_0 is. Call this new approximation x_1, that is,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{2.21}$$

Then, we should have $|x_1 - x^*| < |x_0 - x^*|$. At this point, we can repeat the process in the same manner using the iterative rule

$$x_{n+1} = \Phi(x_n) = x_n + \frac{f(x_n)}{f'(x_n)}, \qquad n > 1, \tag{2.22}$$

to obtain a series of approximations $x_2, x_3, ..., x_n$ approaching the root x^*.

To make sure that the Newton-Raphson iteration function leads to a convergent evaluation of the root(s) of $f(x)$, we need to verify Conditions 1 and 2 that were stated before.

Verification of Condition 1: Recall that $f(x^*) = 0$, so that

$$\Phi(x^*) = x^* - \frac{f(x^*)}{f'(x^*)} = x^*, \tag{2.23}$$

provided that $f'(x) \neq 0$ at x^*.

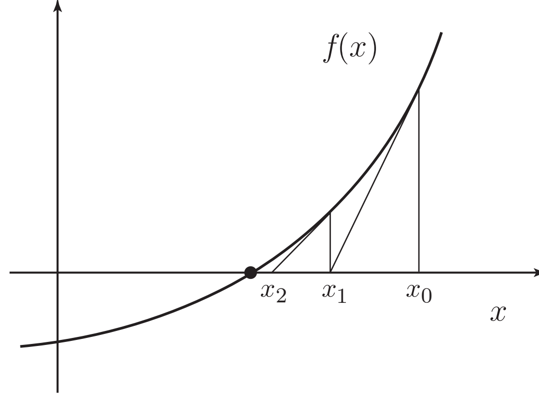

Figure 2.2: Illustration of the Newton-Raphson method.

Verification of Condition 2: The first derivative of $\Phi(x)$ is

$$\Phi'(x) = 1 - \frac{f'(x)}{f'(x)} + \frac{f(x)f''(x)}{f'^2(x)}. \tag{2.24}$$

Provided again that $f'(x^*) \neq 0$, and that $f''(x^*)$ exists, we then have

$$|\Phi'(x^*)| = \left| \frac{f(x^*)f''(x^*)}{f'^2(x^*)} \right| = 0 < 1. \tag{2.25}$$

Moreover, it can be verified that $|\Phi'(x)| < 1$ in the vicinity of x^*, so that δ_n is decreasing as x_n gets close to x^*.

Having verified Conditions 1 and 2, we conclude that $x_n = \Phi(x_{n-1}) \to x^*$ as $n \to \infty$ starting with any initial value x_0 in the vicinity of x^*. In other words, the Newton-Raphson map, defined in Eqn. (2.17) or Eqn. (2.22), leads to a convergent estimation of x^*.

Example 2.4.1. The Newton-Raphson map corresponding to the $f(x)$ considered in Example 2.2.1 is

$$\Phi(x) = x - \frac{f(x)}{f'(x)} = x - \frac{\cos(x) - x}{-\sin(x) - 1} = x + \frac{\cos(x) - x}{\sin(x) + 1}. \tag{2.26}$$

Summary of the method

Given the equation $f(x) = 0$ to solve, the Newton-Raphson method is a fixed-point iteration method based on the general iteration function displayed in Eqn. (2.17). It is a convergent iteration method provided that $f'(x^*) \neq 0$ and $f''(x^*) < \infty$.

Maple code 2.4.1: Newton-Raphson method

```
NRsolve:=proc(f::procedure,x0::realcons,tol::realcons)
   local i,imax,x,xp,abserr,df,Phi;
   imax:=10^6;
   df:=D(f);
   Phi:=x-> x-f(x)/df(x);
   x:=x0;
   abserr:=abs(x-Phi(x));
   for i while abserr>tol do
      xp:=x;
      x:=Phi(x);
      abserr:=abs(x-xp);
      if i>imax then
         error "Too many iterations!";
      end if;
   end do;
   x;
end proc:
```

2.5. Variants of the Newton-Raphson method

Numerical estimation of the derivative

The Newton-Raphson method requires that we evaluate the derivative of $f(x_n)$. This can be difficult numerically or even tiresome, and to circumvent this problem we may use the fact that the distance between the iterate x_{n-1} and the subsequent iterate x_n becomes infinitesimally small as $n \to \infty$ (remember that x_n converges to x^* in that limit). This suggests the following approximation for the derivative:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}. \tag{2.27}$$

Comparing this approximation with the actual definition of the derivative:

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}, \tag{2.28}$$

it should be clear that our approximation of $f'(x_n)$ becomes exact as $x_n \to \infty$, since, in that limit, $|x_n - x_{n-1}| \to 0$.

The Newton-Raphson formula with our approximation becomes

$$x_{n+1} = \Phi(x_n, x_{n-1}) = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}. \tag{2.29}$$

The fixed-point iteration method based on this iteration function is often referred to as the **secant method**. Notice that the iteration function now involves x_n and x_{n-1} and not just x_n.

Root suppression

Suppose that you have calculated a root c of $f(x)$ and attempt to calculate another root of that function. How can you be sure that you will not recalculate c again? One way is by **suppressing** the root c from $f(x)$ by factorizing it from $f(x)$:

$$f(x) = (x - c)g(x). \tag{2.30}$$

Then work at the level of $g(x)$, that is to say, find the roots of

$$g(x) = \frac{f(x)}{x - c}. \tag{2.31}$$

$g(x)$ must have the same roots as $f(x)$, except for c, which has been factorized out of f. To be sure, let d be a second root of $f(x)$. Then,

$$g(d) = \frac{\overbrace{f(d)}^{=0}}{\underbrace{d-c}_{\neq 0}} = 0. \tag{2.32}$$

Thus d is also a root of $g(x)$.

Closely spaced roots

The accuracy used for calculating roots must be set correctly if you suspect that $f(x)$ has one or many roots that are close to one another. Suppose, for example, that $f(x)$ has two roots x_1^* and x_2^* such that $|x_1^* - x_2^*| < d$. Then an iterative method that stops when the absolute error $|x_n - x_{n-1}|$ is smaller than some accuracy ε will be able to distinguish the two roots only if $d > \varepsilon$.

To fix this problem, one must decrease the error or, equivalently, increase the accuracy with which the roots are calculated (for example, by increasing `Digits`).

Complex roots

The Newton-Raphson method can be used to find **complex roots**, that is, roots that are complex numbers. The iteration function to use in this case is the same as for real numbers, namely

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}, \tag{2.33}$$

but z is now a complex number of the form $z = x + iy$, $x, y \in \mathbb{R}$.

The convergence of z_n to some complex root z^* can be measured with the usual Euclidean norm

$$|z_n - z^*| = \sqrt{(x_n - x^*)^2 + (y_n - y^*)^2}. \tag{2.34}$$

Remark 2.5.1. The imaginary unit i is represented by the protected symbol `I` in Maple. Thus, `z:=x+y*I` is a correct complex number. The real and Imaginary parts of that number are given by `Re(z)` and `Im(z)`, respectively. Entering `exp(z)` forces the evaluation of the exponential function for the complex number `z`.

2.6. Iteration method for systems of many equations

Summary of vector notation

A system of equations of the form

$$\begin{aligned} f_1(x_1, x_2, ..., x_n) &= 0 \\ f_2(x_1, x_2, ..., x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, ..., x_n) &= 0 \end{aligned} \tag{2.35}$$

is compactly represented, in vector notation, as

$$\boldsymbol{f}(\boldsymbol{x}) = 0 \tag{2.36}$$

(symbols in bold represent vectors). $\boldsymbol{f} = (f_1, f_2, ..., f_n)$ is the vector of functions, whereas $\boldsymbol{x} = (x_1, x_2, ..., x_n)$ is the vector of variables.

Taylor series in vector notation have the form (here shown up to the linear term)

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}_0) + \frac{\partial \boldsymbol{f}(\boldsymbol{x}_0)}{\partial \boldsymbol{x}}(\boldsymbol{x} - \boldsymbol{x}_0) + \cdots. \tag{2.37}$$

The term $\partial \boldsymbol{f}(\boldsymbol{x}_0)/\partial \boldsymbol{x}$ stands for the matrix of partial derivatives of $\boldsymbol{f}(\boldsymbol{x})$, also called the **Jacobian matrix** of $\boldsymbol{f}(\boldsymbol{x})$:

$$\frac{\partial \boldsymbol{f}(\boldsymbol{x}_0)}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}. \tag{2.38}$$

▶ Examples in class.

This matrix will also be denoted by $D\boldsymbol{f}(\boldsymbol{x})$.

Remark 2.6.1. Maple can handle vectors and vector functions. Remember that column vectors are entered with the syntax `<a,b,c>`, whereas row vectors are entered with the syntax `<a|b|c>`. Building on this, matrices are entered with the syntax `<<a|b|c>,<d|e|f>,<g|h|i>>` (stack of rows).

Fixed-point method in many-dimensions

Given the equation $\boldsymbol{f}(\boldsymbol{x}) = 0$ to solve, find a function $\boldsymbol{\Phi}(\boldsymbol{x})$ such

$$\boldsymbol{x}_n = \boldsymbol{\Phi}(\boldsymbol{x}_{n-1}) \to \boldsymbol{x}^* \qquad \text{as} \quad n \to \infty. \tag{2.39}$$

One possible choice for $\boldsymbol{\Phi}(\boldsymbol{x})$ is $\boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{x}$.

The notation $\boldsymbol{x}_n \to \boldsymbol{x}^*$ means as before that the nth iterate $\boldsymbol{x}_n$ approaches the root $\boldsymbol{x}^*$ as $n \to \infty$. Since we are working in an n-dimensional space, and not just in a one-dimensional space as before, we need to be more precise as to how $\boldsymbol{x}_n$ approaches $\boldsymbol{x}^*$ ($\boldsymbol{x}_n$ can approach $\boldsymbol{x}^*$ from many directions and in many ways). This is done via the notion of **norm**.

We say that $\boldsymbol{x}_n$ approaches $\boldsymbol{x}^*$ as $n \to \infty$ with respect to the norm $\|\cdot\|$ if

$$\|\boldsymbol{x}_n - \boldsymbol{x}^*\| \to 0 \qquad \text{as} \quad n \to \infty. \tag{2.40}$$

There are many definitions of norm; here are some examples:

- Euclidean norm:

$$\|\boldsymbol{a} - \boldsymbol{b}\|_2 = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2} \tag{2.41}$$

- Absolute-value norm:

$$\|\boldsymbol{a} - \boldsymbol{b}\|_1 = \sum_{i=1}^{n} |a_i - b_i| \tag{2.42}$$

- Max or infinite norm:

$$\|\boldsymbol{a} - \boldsymbol{b}\|_\infty = \max_i |a_i - b_i|. \tag{2.43}$$

Convergence of the fixed-point iteration method

To make sure that the sequence of iterates $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n$ generated from $\boldsymbol{\Phi}(\boldsymbol{x})$ actually converges to the root $\boldsymbol{x}^*$, $\boldsymbol{\Phi}(\boldsymbol{x})$ must verify two conditions that are the vector equivalent of Conditions 1 and 2 stated before:

Condition 1: (Fixed-point condition). The root $\boldsymbol{x}^*$ must be a fixed-point of $\boldsymbol{\Phi}(\boldsymbol{x})$, that is, $\boldsymbol{\Phi}(\boldsymbol{x}^*) = \boldsymbol{x}^*$.

Condition 2: (Stability condition). The norm of the Jacobian matrix of $\boldsymbol{\Phi}$, evaluated at the fixed-point $\boldsymbol{x}^*$, must be smaller than unity:

$$\|D\boldsymbol{\Phi}(\boldsymbol{x}^*)\| < 1. \tag{2.44}$$

As before, we need to specify the definition of the norm in the second condition (in this case a matrix norm).

Proof of Condition 2: We repeat with a given choice of norm $\|\cdot\|$ the proof of Condition 2 that we presented for the one-dimensional case (one function of one variable). This yields

$$\begin{aligned}
\delta_n &= \|\boldsymbol{x}_n - \boldsymbol{x}^*\| \\
&= \|\boldsymbol{\Phi}(\boldsymbol{x}_{n-1}) - \boldsymbol{\Phi}(\boldsymbol{x}^*)\|, \qquad \boldsymbol{\Phi}(\boldsymbol{x}_{n-1}) = \boldsymbol{x}_n,\, \boldsymbol{\Phi}(\boldsymbol{x}^*) = \boldsymbol{x}^* \\
&\approx \left\| \boldsymbol{\Phi}(\boldsymbol{x}^*) + \left.\frac{\partial \boldsymbol{\Phi}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}^*} (\boldsymbol{x}_{n-1} - \boldsymbol{x}^*) - \boldsymbol{\Phi}(\boldsymbol{x}^*) \right\| \\
&= \|D\boldsymbol{\Phi}(\boldsymbol{x}^*)(\boldsymbol{x}_{n-1} - \boldsymbol{x}^*)\|, \qquad D = \partial/\partial \boldsymbol{x} \\
&= \|D\boldsymbol{\Phi}(\boldsymbol{x}^*)\|\ \|(\boldsymbol{x}_{n-1} - \boldsymbol{x}^*)\| \\
&= \|D\boldsymbol{\Phi}(\boldsymbol{x}^*)\|\ \delta_{n-1}.
\end{aligned} \tag{2.45}$$

Therefore, if we can find a norm such that $\|D\boldsymbol{\Phi}(\boldsymbol{x}^*)\| < 1$, then $\delta_n < \delta_{n-1}$.

In practice, it is convenient to replace the matrix norm in Condition 2 by the **spectral radius** of $D\boldsymbol{\Phi}(x^*)$, defined as

$$\rho\,(D\boldsymbol{\Phi}(\boldsymbol{x})) = \max_i |\lambda_i|, \tag{2.46}$$

where λ_i represents the eigenvalues of the Jacobian matrix of $\boldsymbol{\Phi}$ evaluated at $\boldsymbol{x}$. The spectral radius is not a norm; still, it can be used to decide whether $\boldsymbol{\Phi}$ is converging or not. This is proved in Section 2.8 at the end of this chapter. With the spectral radius, we then have the following stability condition:

Condition 2′: If $\rho\,(D\boldsymbol{\Phi}(\boldsymbol{x}^*)) < 1$, then $\boldsymbol{x}^*$ is an attracting fixed-point of $\boldsymbol{\Phi}$.

2.7. Newton-Raphson method in many dimensions

The n-dimensional generalization of the Newton-Raphson iteration function has the form

$$\boldsymbol{\Phi}(\boldsymbol{x}) = \boldsymbol{x} - [D\boldsymbol{f}(\boldsymbol{x})]^{-1}\boldsymbol{f}(\boldsymbol{x}). \tag{2.47}$$

$[D\boldsymbol{f}(\boldsymbol{x})]^{-1}$ is the inverse of the Jacobian matrix of $\boldsymbol{f}(\boldsymbol{x})$.

To justify this form of $\boldsymbol{\Phi}(\boldsymbol{x})$, we need only to translate our previous derivation of the Newton-Raphson method in vector notation. First use the expression of the n-dimensional Taylor series found in Eqn. (2.37) to write

$$\boldsymbol{f}(\boldsymbol{x}^*) = 0 \approx \boldsymbol{f}(\boldsymbol{x}_0) + D\boldsymbol{f}(\boldsymbol{x}_0)(\boldsymbol{x}^* - \boldsymbol{x}_0). \tag{2.48}$$

This is the n-dimension generalization of Eqn. (2.19). Then, solving for $\boldsymbol{x}^*$ we find

$$\boldsymbol{x}^* \approx \boldsymbol{x}_1 = \boldsymbol{x}_0 - [D\boldsymbol{f}(\boldsymbol{x}_0)]^{-1}\boldsymbol{f}(\boldsymbol{x}_0) \tag{2.49}$$

as the n-dimensional generalization of Eqn. (2.20).

Note that in solving Eqn. (2.48) for $\boldsymbol{x}^*$, you cannot write

$$\boldsymbol{x}^* \approx \boldsymbol{x}_1 = \boldsymbol{x}_0 - \frac{\boldsymbol{f}(\boldsymbol{x}_0)}{D\boldsymbol{f}(\boldsymbol{x}_0)}. \tag{2.50}$$

You cannot divide a vector by a matrix! Remember that the proper way to move a matrix from one side of an equation to the other is by multiplying the matrix by its inverse.

Remark 2.7.1. In Maple, the product of a matrix A with a vector $\boldsymbol{x}$ is denoted by `A.x`. The dot "`.`" is the product operator.

Maple code 2.7.1: Newton-Raphson method for 3D vectors

```
use LinearAlgebra in
   NR3solve:=proc(f::procedure,x0::Vector3,tol::realcons)
      local i,imax,x,xp,abserr,Df,Phi;
      imax:=10^6;
      Df:=unapply(VectorCalculus[Jacobian](f(x),
            [x[1],x[2],x[3]]),x::Vector3);
      Phi:=(x::Vector3)->x-Df(x)^(-1).f(x);
      x:=x0;
      abserr:=VectorNorm(x-Phi(x));
      for i while abserr>tol do
         xp:=x;
         x:=Phi(x);
         abserr:=VectorNorm(x-xp);
         if i>imax then
            error "Too many iterations!";
         end if;
      end do;
      x;
   end proc:
end use:
```

The previous code uses the object `Vector3`, a vector of three components, defined with the line

```
>TypeTools[AddType](Vector3,{'Vector'(3),symbol});
```

The code also uses the package `LinearAlgebra`, invoked with the command `use`.

2.8. *Stability condition using the spectral norm

Repeating the error analysis that we carried out for the Newton-Raphson method, we can write

$$(\boldsymbol{x}_{i+1} - \boldsymbol{x}^*) = D\boldsymbol{\Phi}(\boldsymbol{x}^*)(\boldsymbol{x}_i - \boldsymbol{x}^*), \tag{2.51}$$

where $D\boldsymbol{\Phi}(\boldsymbol{x}^*)$ is the Jacobian of $\boldsymbol{\Phi}$ evaluated at the root (fixed-point) $\boldsymbol{x}^*$. Let $\boldsymbol{e} = \boldsymbol{x}_i - \boldsymbol{x}^*$. Then, sufficiently close to $\boldsymbol{x}^*$, we have

$$\boldsymbol{e}_{i+1} = D\boldsymbol{\Phi}(\boldsymbol{x}^*)\boldsymbol{e}_i \tag{2.52}$$

and, by induction,

$$\boldsymbol{e}_n = [D\boldsymbol{\Phi}(\boldsymbol{x}^*)]^n \boldsymbol{e}_0. \tag{2.53}$$

Convergence of the iteration requires that $\boldsymbol{e}_n$ tends to the zero vector as $n \to \infty$ for all (sufficiently small) initial vector $\boldsymbol{e}_0$. In particular, suppose $\boldsymbol{e}_0$ is an eigenvector of the matrix $D\boldsymbol{\Phi}(\boldsymbol{x}^*)$ with corresponding eigenvalue λ, then

$$\boldsymbol{e}_n = [D\boldsymbol{\Phi}(\boldsymbol{x}^*)]^n \boldsymbol{e}_0 = \lambda^n \boldsymbol{e}_0. \tag{2.54}$$

Therefore, we see that $\boldsymbol{e}_n \to 0$ as $n \to \infty$ if and only if $|\lambda| < 1$. As this must hold for any initial vector $\boldsymbol{e}_0$, we must require that all the eigenvalues of $D\boldsymbol{\Phi}(x^*)$ be smaller than 1 in magnitude.

Further reading

Sections 2.1-2.3, 9.1, 9.2 of R.L. Burden, J.D. Faires, Numerical Analysis, Prindle, Weber & Schmidt, 1985.

Chapter 3

Numerical computation of eigenvalues and eigenvectors

3.1. Short revision of matrix algebra

Let A be a square, real matrix (that is, with real elements) of dimension $n \times n$. The **eigenvalues** and **eigenvectors** of A are defined by the equation

$$A\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i, \quad i = 1, 2, \ldots, n. \tag{3.1}$$

λ_i is the ith **eigenvalue** of A with corresponding **eigenvector** $\boldsymbol{x}_i$. Note that vectors are written in bold script.

One can always order the eigenvalues in decreasing order of magnitude:

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|. \tag{3.2}$$

We shall always use λ_1 to denote the **maximum** or **dominant eigenvalue** of A. Some eigenvalues may have the same value; we say in this case that they are **degenerate**. The **multiplicity** of an eigenvalue is the number of times a given value repeats itself in the sequence of eigenvalues.

The eigenvalues of a matrix A are given by the roots of the **characteristic equation** or **characteristic polynomial**, given by

$$\det(A - \lambda \mathbb{I}) = 0, \tag{3.3}$$

where $\mathbb{I}$ denotes the identity matrix.

Example 3.1.1. Consider

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}. \tag{3.4}$$

The matrix $A - \lambda\mathbb{I}$ is

$$A - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1-\lambda & 2 \\ 2 & 1-\lambda \end{pmatrix}. \tag{3.5}$$

The determinant of this matrix is

$$\det(A - \lambda\mathbb{I}) = \begin{vmatrix} 1-\lambda & 2 \\ 2 & 1-\lambda \end{vmatrix} = (1-\lambda)^2 - 4. \tag{3.6}$$

Solving $\det(A - \lambda\mathbb{I}) = 0$ for λ yields the two eigenvalues of A, namely $\lambda_1 = 3$ and $\lambda_2 = -1$.

The calculation of the eigenvalues from the characteristic equation is generally not efficient in terms of running time and numerical errors (convince yourself by considering an $n \times n$ matrix). The goal of this chapter is to study more efficient methods.

If A is real, then the eigenvalues of A are real or come in conjugate pairs ($\lambda^* = \lambda$).

Let A be a real, square matrix. If the eigenvalues of A are all distinct, then its eigenvectors are **linearly independent**.

A real, square matrix A which is **symmetric** (i.e., $A^T = A$) has n **distinct** eigenvalues. Therefore, its eigenvectors are linearly independent.

A set $\{\boldsymbol{x}_i\}_{i=1}^n$ of n linearly independent eigenvectors can be used to define a **complete basis** for an n-dimensional linear space L^n. This means that any vector $\boldsymbol{v} \in L^n$ can be **decomposed** in the basis $\{\boldsymbol{x}_i\}_{i=1}^n$ as

$$\boldsymbol{v} = \sum_{i=1}^{n} a_i \boldsymbol{x}_i. \tag{3.7}$$

The coefficients $a_i = \boldsymbol{v}^T \boldsymbol{x}_i \in \mathbb{R}$ are the **projections** of $\boldsymbol{v}$ onto each of the eigenvectors $\boldsymbol{x}_i$.

If $\boldsymbol{x}_i$ is an eigenvector of A, then so is $\boldsymbol{x}_i' = c\boldsymbol{x}_i$, where c is any constant. Indeed,

$$A(c\boldsymbol{x}_i) = c(A\boldsymbol{x}_i) = c\lambda_i \boldsymbol{x}_i = \lambda_i(c\boldsymbol{x}_i). \tag{3.8}$$

This shows that eigenvectors are defined up to a multiplicative constant. In other words, if $\boldsymbol{x}$ is an eigenvector of A and $\boldsymbol{v}$ is a multiple of $\boldsymbol{x}$, then $\boldsymbol{v}$ is also an eigenvector of A.

A **similarity transform** is a transform of the form

$$X^{-1}AX = A' \tag{3.9}$$

such that the eigenvalues of A' are the same as those of A. In that sense, we say that a similarity transform **preserves** the eigenvalues of A.

To **diagonalise** a matrix A means to apply a similarity transform to A so as to obtain a **diagonal matrix** Λ:

$$X^{-1}AX = \Lambda = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}. \tag{3.10}$$

The diagonal elements of Λ are the eigenvalues of A because similarity transforms leave the eigenvalues of a matrix unchanged. The columns of X are the eigenvectors of A.

An **orthogonal** matrix P is a matrix having the property that $P^{-1} = P^T$.

A QR **decomposition** of A is a matrix decomposition of the form $A = QR$, where Q is **orthogonal** and R is **upper triangular**, i.e., contains only zeros below the diagonal.

Given the QR decomposition $A = QR$, we define the QR **transformation** of A as

$$A' = Q^T AQ. \tag{3.11}$$

Since $Q^T A = R$, this is equivalent to

$$A' = RQ. \tag{3.12}$$

Thus the QR transformation of A is simply obtained by multiplying the two factors of the QR decomposition in opposite order.

A QR transformation is clearly a similarity transform, so that the eigenvalues and eigenvectors of A' are the same as those of A.

Remark 3.1.1. The eigenvalues and eigenvectors of a matrix can be found in Maple with the procedure `Eigenvectors` (part of the `LinearAlgebra` package). To obtain only the eigenvalues, use the procedure `Eigenvalues`. `QRDecomposition`, which is also part of the `LinearAlgebra` package, returns the QR decomposition of a matrix.

3.2. Power method

The **power method** finds the largest (in magnitude) eigenvalue of a matrix and its corresponding eigenvector.

We consider, as always, a real matrix A of dimension $n \times n$. We assume that A has n distinct eigenvalues, which we rank in decreasing order of magnitude:

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|. \tag{3.13}$$

The first eigenvalue λ_1 in the ranking is called the **dominant eigenvalue** of A.

The fact that A has n distinct eigenvalues means, as was mentioned, that the corresponding eigenvectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ form a complete basis for n-dimensional vectors. This means that any arbitrary vector $\boldsymbol{v} \in \mathbb{R}^n$ can be written in the form

$$\boldsymbol{v} = \sum_{i=1}^{n} a_i \boldsymbol{x}_i. \tag{3.14}$$

Pre-multiplying this equation by A yields

$$A\boldsymbol{v} = \sum_{i=1}^{n} a_i A\boldsymbol{x}_i = \sum_{i=1}^{n} a_i \lambda_i \boldsymbol{x}_i, \tag{3.15}$$

since $\boldsymbol{x}_i$ is an eigenvector of A with corresponding eigenvalue λ_i. Repeating this operation k times, we obtain

$$A^k \boldsymbol{v} = \underbrace{AA \cdots A}_{k \text{ times}} \boldsymbol{v} = \sum_{i=1}^{n} a_i \lambda_i^k \boldsymbol{x}_i. \tag{3.16}$$

At this point, observe that the largest term of the sum is the first one ($i = 1$), since λ_1 is the dominant eigenvalue. To make this more obvious, factorise λ_1 out of the sum:

$$A^k \boldsymbol{v} = \lambda_1^k \sum_{i=1}^{n} a_i \left(\frac{\lambda_i}{\lambda_1} \right)^k \boldsymbol{x}_i. \tag{3.17}$$

In the limit where $k \to \infty$, we have

$$\lim_{k \to \infty} \left(\frac{\lambda_i}{\lambda_1} \right)^k = 0, \tag{3.18}$$

for all $i \neq 1$, since again λ_1 is the dominant eigenvalue. Therefore,

$$\lim_{k \to \infty} A^k \boldsymbol{v} = a_1 \lambda_1^k \boldsymbol{x}_1. \tag{3.19}$$

This result holds obviously provided that $a_1 \neq 0$. This limit suggests the following algorithm for finding λ_1 and $\boldsymbol{x}_1$:

Power method algorithm

Step 1: (Initialisation) Choose an arbitrary vector $\boldsymbol{v}_0$.

Step 2: (Iteration) Compute $\boldsymbol{v}_k = A^k \boldsymbol{v}_0$ iteratively using $\boldsymbol{v}_k = A\boldsymbol{v}_{k-1}$.

Step 3: (Stop iteration) For k large enough, $\boldsymbol{v}_k$ should be "close" to $\boldsymbol{x}_1$ because of Eqn. (3.19). By "close," we mean that $\boldsymbol{v}_k$ is nearly aligned with $\boldsymbol{x}_1$.

Step 4: (Calculation of $\boldsymbol{x}_1$) $\boldsymbol{v}_k$ is the approximation of $\boldsymbol{x}_1$ up to some arbitrary constant. We write this as $\boldsymbol{v}_k \approx c\boldsymbol{x}_1$.

Step 5: (Calculation of λ_1) Since $\boldsymbol{v}_k \approx c\boldsymbol{x}_1$, with c a constant, we also have

$$A\boldsymbol{v}_k = \boldsymbol{v}_{k+1} \approx \lambda_1 \boldsymbol{v}_k. \tag{3.20}$$

The last approximation holds for any component of the vector $\boldsymbol{v}_k$, so that the approximation of λ_1, which we call $\lambda^{(k)}$, can be calculated as

$$\lambda^{(k)} = \frac{\boldsymbol{v}_{k+1}[i]}{\boldsymbol{v}_k[i]} = \frac{(A\boldsymbol{v}_k)[i]}{\boldsymbol{v}_k[i]}. \tag{3.21}$$

In this expression, $\boldsymbol{v}[i]$ means the ith component of $\boldsymbol{v}$.

Stopping rule

The power method algorithm can be stopped after some number k of iterations (one iteration = one matrix multiplication). A better stopping rule is to stop when the iteration error is below some threshold ε, i.e.,

$$\delta\boldsymbol{v}_k \approx \|\boldsymbol{v}_k - \boldsymbol{v}_{k-1}\| < \varepsilon. \tag{3.22}$$

The norm is arbitrary. You can choose it, for example, to be the normal Euclidean norm

$$||\boldsymbol{y}||_2 = \sqrt{\sum_{i=1}^{n} \boldsymbol{y}[i]} \tag{3.23}$$

or the max (or infinity norm)

$$||\boldsymbol{y}||_\infty = \max_i \ \boldsymbol{y}[i]. \tag{3.24}$$

Yet another stopping rule can be imposed by requiring that

$$\delta\lambda^{(k)} \approx |\lambda^{(k)} - \lambda^{(k-1)}| < \varepsilon. \tag{3.25}$$

This is an acceptable stopping rule if one is only interested in obtaining the dominant eigenvalue and not its corresponding eigenvector.

Normalisation step

We know from Eqn. (3.19) that

$$A^k \boldsymbol{v} \to \lambda_1^k a_1 \boldsymbol{x}_1, \quad \text{as} \quad k \to \infty. \tag{3.26}$$

There is a problem with this limit. If $|\lambda_1| > 1$, then the vector obtained from $A^k \boldsymbol{v}$ has a norm "blowing up" to infinity, whereas if $|\lambda_1| < 1$, then $A^k \boldsymbol{v} \to 0$.

To contain this problem, we can use the fact that eigenvectors can be multiplied by a constant without changing the fact that they are eigenvectors. With this in mind, we can divide the vector $\boldsymbol{v}_k$ at each iteration step of the power method by its maximum component (i.e., its max norm) to make sure that the norm of that vector does not go to infinity or zero as $k \to \infty$. This **normalisation step** can be implemented in practice as an added step in Step 2 above:

Step 2′: (Normalisation) Find the maximum component of $\boldsymbol{v}_k$ in magnitude, then divide all the components of $\boldsymbol{v}_k$ by this maximum component. The resulting **normalised** vector is the iterate $\boldsymbol{v}_k$.

With the normalisation step, the approximation $\lambda^{(k)}$ to λ_1 is given by $\lambda^{(k)} = (A\boldsymbol{v}_k)[p]$, where p is the index of the maximum-magnitude component of $\boldsymbol{v}_k$. This simply follows from Eqn. (3.21) and the fact that $\boldsymbol{v}_k[p] = 1$ by definition of p.

Choice of the initial vector

The first step of the power method is to choose an arbitrary vector $\boldsymbol{v}_0$. If by any chance the choice of $\boldsymbol{v}_0$ is such that $a_1 = 0$, then $A^k \boldsymbol{v}_0 \not\to \boldsymbol{x}_1$. In this case, one should choose a different initial vector.

The next code treats the initial vector as an optional argument (using `nargs`), and includes an error message displayed if a zero eigenvalue is found.

3.3. Deflation method

Deflation is a way of removing the dominant eigenvalue λ_1 from the set of eigenvalues of A, so that the next dominant eigenvalue can be calculated using the power method again. Using this method repeatedly, one can compute all the eigenvalues of A.

Maple code 3.2.1: Power method (basic version)

```
use LinearAlgebra in
   PW:=proc(A::Matrix,tol::realcons,x0::Vector)
      local x,p,i,y,lambda,abserr,imax;
      imax:=10^4;
      p:=maxind(x);
      x:=x/x[p];
      for i to imax do
         y:=A.x;
         lambda:=y[p];
         p:=maxind(y);
         y:=y/y[p];
         abserr:=VectorNorm(x-y,'Euclidean');
         x:=y;
         if abserr<tol then
            return lambda,x;
         end if;
      end do;
      error "Too many iterations!";
   end proc:
end use:
```

Maple code 3.2.2: Power method (full version)

```
use LinearAlgebra in
   PW:=proc(A::Matrix,tol::realcons,x0::Vector)
      local x,p,i,y,lambda,abserr,ndim,imax;
      imax:=10^4;
      ndim:=ColumnDimension(A);
      if nargs>2 then
         x:=x0;
      else
         x:=Vector(ndim,fill=1.0);
      end if;
      p:=maxind(x);
      x:=x/x[p];
      for i to imax do
         y:=A.x;
         lambda:=y[p];
         p:=maxind(y);
         if y[p]=0 then
            error "Zero eigenvalue found.
                Select a different initial vector.";
         end if;
         y:=y/y[p];
         abserr:=VectorNorm(x-y,'Euclidean');
         x:=y;
         if abserr<tol then
            return lambda,x;
         end if;
      end do;
      error "Too many iterations!";
   end proc:
end use:
```

Maple code 3.2.3: `maxind`

```
maxind:=proc(x::Vector)
   # Returns the position of the maximum component of x
   local p,m,pmax,ndim;
   ndim:=LinearAlgebra[Dimension](x);
   pmax:=1;
   m:=0;
   for p to ndim do
      if abs(x[p])>m then
         pmax:=p;
         m:=abs(x[p]);
      end if;
   end do;
   pmax;
end proc:
```

Here is a version of deflation due to **Wielandt** (1944). We assume, as usual, that

$$A\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i, \qquad i = 1, 2, \ldots, n, \tag{3.27}$$

and we want to remove from this system of eigenvalues and eigenvectors the dominant eigenvalue λ_1 with its corresponding eigenvector $\boldsymbol{x}_1$. To do so, we first choose an integer p such that $\boldsymbol{x}_1[p] \neq 0$, and divide all the components of $\boldsymbol{x}_1$ by $\boldsymbol{x}_1[p]$ so that $\boldsymbol{x}_1[p] = 1$. This is a normalisation step that does not change the eigenvector nature of $\boldsymbol{x}_1$.

Next we compute a transformed matrix A_p according to the formula

$$A_p = A - \boldsymbol{x}_1 \boldsymbol{a}_p, \tag{3.28}$$

where a_p is the pth row of A. $\boldsymbol{x}_1$ is a column vector, whereas $\boldsymbol{a}_p$ is a row vector, so that A_p can be visualised as follows:

$$A_p = \overset{A}{\begin{pmatrix} & & \\ & & \\ & & \end{pmatrix}} - \overset{\boldsymbol{x}_1}{\begin{pmatrix} \\ \\ \\ \end{pmatrix}} \overset{\boldsymbol{a}_p}{\begin{pmatrix} & & \end{pmatrix}} \tag{3.29}$$

The pth row of A_p contains only zeros because the pth component of $\boldsymbol{x}_1$ is

1. This is illustrated next:

$$\begin{aligned}
A_p &= \overset{A}{\begin{pmatrix} & & \\ & \boldsymbol{a}_p & \\ & & \end{pmatrix}} - \overset{\boldsymbol{x}_1}{\begin{pmatrix} \\ 1 \\ \\ \end{pmatrix}} \begin{pmatrix} & \boldsymbol{a}_p & \end{pmatrix} \\
&= \overset{A}{\begin{pmatrix} & & \\ & \boldsymbol{a}_p & \\ & & \end{pmatrix}} - \overset{\boldsymbol{x}_1 \boldsymbol{a}_p}{\begin{pmatrix} & & \\ & \boldsymbol{a}_p & \\ & & \end{pmatrix}} \\
&= \begin{pmatrix} & & \\ 0 & 0 \cdots & 0 \\ & & \end{pmatrix}
\end{aligned} \tag{3.30}$$

The claim at this point is that the eigenvalues of A_p are the same as those of A, except that the dominant eigenvalue λ_1 of A is zero for A_p. To prove this, let us pre-multiply the eigenvector $\boldsymbol{x}_i$ with A_p:

$$\begin{aligned}
A_p \boldsymbol{x}_i &= (A - \boldsymbol{x}_1 \boldsymbol{a}_p) \boldsymbol{x}_i \\
&= A \boldsymbol{x}_i - \boldsymbol{x}_1 \boldsymbol{a}_p \boldsymbol{x}_i \\
&= \lambda_i \boldsymbol{x}_i - \boldsymbol{x}_1 \lambda_i \boldsymbol{x}_i[p] \\
&= \lambda_i (\boldsymbol{x}_i - \boldsymbol{x}_i[p] \boldsymbol{x}_1).
\end{aligned} \tag{3.31}$$

For $i = 1$, this yields

$$A_p \boldsymbol{x}_1 = \lambda_1 (\boldsymbol{x}_1 - \underbrace{\boldsymbol{x}_1[p]}_{=1} \boldsymbol{x}_1) = 0. \tag{3.32}$$

Consequently, the first eigenvalue of A_p is zero, as claimed. For $i \neq 1$, we have instead

$$A_p \boldsymbol{x}_i = \lambda_i (\boldsymbol{x}_i - \boldsymbol{x}_i[p] \boldsymbol{x}_1). \tag{3.33}$$

This can be put in a more symmetric form using the fact that $A\boldsymbol{x}_1 = 0$:

$$A_p (\boldsymbol{x}_i - \boldsymbol{x}_i[p] \boldsymbol{x}_1) = \lambda_i (\boldsymbol{x}_i - \boldsymbol{x}_i[p] \boldsymbol{x}_1). \tag{3.34}$$

Defining

$$\boldsymbol{y}_i = \boldsymbol{x}_i - \boldsymbol{x}_i[p] \boldsymbol{x}_1, \tag{3.35}$$

we then obtain

$$A_p \boldsymbol{y}_i = \lambda_i \boldsymbol{y}_i. \tag{3.36}$$

From this last equation, we see that the eigenvalues of A_p for $i \neq 1$ are the same as those of A. To summarize, then, we have

$$\{\lambda_1, \lambda_2, \ldots, \lambda_n\} \tag{3.37}$$

as the set of eigenvalues of A and

$$\{0, \lambda_2, \ldots, \lambda_n\} \tag{3.38}$$

as the set of eigenvalues of A_p. Since λ_1 is not dominant anymore for A_p, we can apply the power method to A_p to obtain the dominant eigenvalue of A_p, namely λ_2, thereby obtaining the **second dominant** eigenvalue of A. Repeating this process iteratively, we can find all the eigenvalues of A.

Example 3.3.1. Consider

$$A = \begin{pmatrix} -4 & -14 & 0 \\ -5 & 13 & 0 \\ -1 & 0 & 2 \end{pmatrix}. \tag{3.39}$$

The eigenvalues of A are $\lambda_1 = 6$, $\lambda_2 = 3$ and $\lambda = 2$. The eigenvector corresponding to these eigenvalues are

$$\boldsymbol{x}_1 = \begin{pmatrix} -4 \\ -20/7 \\ 1 \end{pmatrix}, \qquad \boldsymbol{x}_2 = \begin{pmatrix} -1 \\ -1/2 \\ 1 \end{pmatrix}, \qquad \boldsymbol{x}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{3.40}$$

We take the first eigenvector $\boldsymbol{x}_1$, and normalise it using its first element ($p = 1$), so as to have $\boldsymbol{x}_1[1] = 1$:

$$\boldsymbol{x}_1 = -\frac{1}{4}\begin{pmatrix} -4 \\ -20/7 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 5/7 \\ -1/4 \end{pmatrix}. \tag{3.41}$$

We next calculate $A_p = A - \boldsymbol{x}_1 \boldsymbol{a}_p$ for $p = 1$ using $\boldsymbol{x}_1$ and $\boldsymbol{a}_1$, the first row of A:

$$\begin{aligned} A_1 &= \begin{pmatrix} -4 & -14 & 0 \\ -5 & 13 & 0 \\ -1 & 0 & 2 \end{pmatrix} - \begin{pmatrix} 1 \\ 5/7 \\ -1/4 \end{pmatrix} (-4\ 14\ 0) \\ &= \begin{pmatrix} -4 & -14 & 0 \\ -5 & 13 & 0 \\ -1 & 0 & 2 \end{pmatrix} - \begin{pmatrix} -4 & -14 & 0 \\ -20/7 & 10 & 0 \\ 1 & -7/2 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 0 \\ -15/7 & 3 & 0 \\ -2 & 7/2 & 2 \end{pmatrix}. \end{aligned} \tag{3.42}$$

Note that the first row of A_1 contains only zeros, as expected. The eigenvalues of A_1 are now 3, 2 and 0, which are the eigenvalues of A, except for λ_1 which has been "put down" to 0.

▶ Examples in class.

Eigenvectors of A and A_p

The eigenvalues of A_p are the same as those of A (except for λ_1), but the eigenvectors of A_p are not eigenvectors of A. The eigenvectors of A_p are given by Eqns. (3.34) and (3.36) above:

$$A_p \boldsymbol{y}_i = \lambda_i \boldsymbol{y}_i, \qquad \boldsymbol{y}_i = \boldsymbol{x}_i - \boldsymbol{x}_i[p]\boldsymbol{x}_1. \tag{3.43}$$

In order to find $\boldsymbol{x}_i$ from $\boldsymbol{y}_i$, we need to find $\boldsymbol{x}_i[p]$. Pre-multiplying the definition of $\boldsymbol{y}_i$ by A gives the matrix equation

$$A\boldsymbol{y}_i = A\boldsymbol{x}_i - A\boldsymbol{x}_i[p]\boldsymbol{x}1 = \lambda_i \boldsymbol{x}_i - \boldsymbol{x}_i[p]\lambda_1 \boldsymbol{x}_1, \qquad i \neq 1. \tag{3.44}$$

The pth row if this equation is

$$\boldsymbol{a}_p \boldsymbol{y}_i = \lambda_i \boldsymbol{x}_i[p] - \boldsymbol{x}_i[p]\lambda_1 \underbrace{\boldsymbol{x}_1[p]}_{=1}. \tag{3.45}$$

Solving this scalar equation gives

$$\boldsymbol{x}_i[p] = \frac{\boldsymbol{a}_p \boldsymbol{y}_i}{\lambda_i - \lambda_1}. \tag{3.46}$$

Therefore,

$$\boldsymbol{x}_i = \boldsymbol{y}_i + \frac{\boldsymbol{a}_p \boldsymbol{y}_i}{\lambda_i - \lambda_1}\boldsymbol{x}_1, \qquad i \neq 1. \tag{3.47}$$

Deflation of the matrix A_p

The fact that the pth row of A_p contains only zeros implies that $\boldsymbol{y}_i[p] = 0$, provided that $\lambda_i \neq 0$. In this case, we can forget about the pth component of $\boldsymbol{y}_i$, which means that we can forget about the pth column of A_p, that is, we can delete that column from A_p. We can also delete from A_p the pth row because it contains only zeros.

Deleting both the pth column of A_p and its pth row is called **deflating** A_p, and has the effect of deleting the zero eigenvalue of A_p. By deflating A_p we thus obtain an $(n-1) \times (n-1)$ matrix B whose eigenvalues are

$$\{\lambda_2, \lambda_3 \ldots, \lambda_n\}. \tag{3.48}$$

The eigenvectors of B are the same as those of A_p except for the deleted zero at position p.

Example 3.3.2. Let us consider again the matrix A_1 calculated in the previous example:

$$A_1 = \begin{pmatrix} 0 & 0 & 0 \\ -15/7 & 3 & 0 \\ -2 & 7/2 & 2 \end{pmatrix}. \tag{3.49}$$

Removing the first row and first column of this matrix yields the following deflated matrix:

$$B = \begin{pmatrix} 3 & 0 \\ 7/2 & 2 \end{pmatrix}. \tag{3.50}$$

It is easy to see that two the eigenvalues of B correspond to the two non-zero eigenvalues of A_1, namely 3 and 2. This confirms the fact that deflating A_p has the effect of removing the zero eigenvalue of A_p while leaving the other eigenvalues unchanged.

Recursive call of Wielandt's deflation

The calculation of the dominant eigenvalue of A together with its eigenvector, followed by the calculation of A_p and the deflation of that matrix corresponds to one step of the Wielandt deflation method. It should be clear that by repeating this process, i.e., by repeating Wielandt's deflation again and again, we can obtain all the eigenvalues of A from the results of each deflation step. Using Eqn. (3.47), we can also compute all the eigenvectors of A. Here is how we do it:

Step 1: (Power method) Compute the dominant eigenvalue λ_1 of A and its corresponding eigenvector $\boldsymbol{x}_1$ using the power method.

Step 2: (Normalisation) Find p such that $|\boldsymbol{x}_1[p]|$ is maximal. Normalize $\boldsymbol{x}_1$ with this component, i.e., `x:=x/x[p]`.

Step 3: (Transformed Wielandt's matrix) Compute

$$A_p = A - \boldsymbol{x}_1 \boldsymbol{a}_p, \tag{3.51}$$

where $\boldsymbol{a}_p$ is the pth column of A.

Step 4: (Deflation) Remove the pth column and the pth row of A_p.

Step 5: (Recursive application of Wielandt's deflation) Re-apply Steps 1-4 to the deflated matrix. Each recursive call decreases the dimension of the deflated matrix by 1 because of deflation step.

Step 6: (Stopping rule) Stop repeating Step 5 when a 1×1 matrix is reached. The last eigenvalue of A is the single element of that 1×1 matrix; its eigenvector is any 1×1 vector. At this point, you will have computed all the eigenvalues of A, and the eigenvectors of the sub-deflated matrices.

Step 7: (Calculation of the eigenvectors). Starting with the 1×1 eigenvector of the last deflated matrix, obtain with Eqn. (3.47) the two eigenvectors of the previous "inflated" matrix. Make sure in doing this calculation that you inflate zeros in the eigenvector at the positions at which you have deflated the previous matrix. Repeat this process iteratively to all the sub-deflated matrices to recover in the end all the eigenvectors of A.

Here is a schematic representation of the recursive Wielandt deflation algorithm:

$$\begin{array}{llcl}
(n\times n) & & A & \rightarrow \lambda_1, \boldsymbol{x}_1 \\
& \text{Transform} & \downarrow & \\
& & A_p & \\
& \text{Deflate} & \downarrow & \\
(n-1)\times(n-1) & & B & \rightarrow \lambda_2, \boldsymbol{y}_2 \\
& \text{Transform} & \downarrow & \\
& & B_p & \\
& \text{Deflate} & \downarrow & \\
(n-2)\times(n-2) & & C & \\
& & \downarrow & \\
& & \vdots & \\
& & \downarrow & \\
1\times 1 & & D & \rightarrow \lambda_n = d, \boldsymbol{z}_n = (1)
\end{array} \tag{3.52}$$

After having reached the 1×1 matrix, the eigenvectors $\boldsymbol{x}_i$ for $i = 2, 3, \ldots, n$ are calculated by inflating the system back to its original size, that is, by climbing up the diagram from the 1×1 matrix D back to the original $n\times n$ matrix A. Note again that the last eigenvalue λ_n is equal to the single element of the matrix D. Its corresponding eigenvector is a one-dimensional vector, which we can take to be the unit vector (1) (<1> in Maple notation).

A complete example of Wielandt's algorithm is included in a complement available on the course's webpage.

3.4. QR algorithm

We can diagonalise a matrix A by iteratively applying QR transformations on A. The result of the iteration will converge according to one of the two following results:

1. For a general, real matrix A, the sequence of QR transformations will transform A to an upper triangular matrix U. In this case, the eigenvalues of A are given by the diagonal elements of U.

Maple code 3.3.1: Wielandt deflation algorithm (eigenvalues only)

```
use LinearAlgebra in
   WDEF1:=proc(A::Matrix,tol::realcons)
      local i,Ap,x,p,ap,lambda,S;
      S:=NULL;
      Ap:=A;
      for i to ColumnDimension(Ap)-1 do
         lambda,x:=PW(Ap,tol);
         S:=S,lambda;
         p:=maxind(x);
         x:=x/x[p];
         ap:=Row(Ap,p);
         Ap:=Ap-x.ap;
         Ap:=SubMatrix(Ap,[1..p-1,p+1..-1],[1..p-1,p+1..-1]);
      end do;
      S:=S,Ap[1,1];
      return [S];
   end proc;
end use:
```

Maple code 3.3.2: Wielandt deflation algorithm (eigenvalues and eigenvectors)

```
use LinearAlgebra in
   WDEF2:=proc(A::Matrix,tol::float)
      local lambda1,x1,p,ap,Ap,S,ev,yi;
      if ColumnDimension(A)=1 then
         return [A[1,1],<1>];
      end if;
      lambda1,x1:=PW(A,tol);
      p:=maxind(x1);
      x1:=x1/x1[p];
      ap:=Row(A,p);
      Ap:=A-x1.ap;
      Ap:=SubMatrix(Ap,[1..p-1,p+1..-1],[1..p-1,p+1..-1]);
      S:=[lambda1,x1];
      for ev in [WDEF2(Ap,tol)] do
         yi:=ev[2];
         yi:=Vector([SubVector(yi,1..p-1),0,SubVector(yi,p..-1)]);
         S:=S,[ev[1],yi+(ap.yi)/(ev[1]-lambda1)*x1];
      end do;
      S;
   end proc;
end use:
```

2. If A is symmetric, then the sequence of QR transformations will converge to a diagonal form Λ. In this case, the eigenvalues of A are given by the diagonal elements of Λ.

If A is symmetric, there is also a way to obtain the eigenvectors of A, not just the eigenvalues, using the sequence of QR transformations. This is described in the next algorithm.

QR diagonalisation algorithm

Step 1: (Initialisation) Set $A_0 = A$ as the initial matrix of the iterative process.

Step 2: (QR decomposition) Find the QR decomposition of A_0

$$A_0 = Q_0 R_0. \tag{3.53}$$

Step 3: (QR transformation) Calculate $A_1 = Q_0^T A_0 Q_0$. Following Eqn. (3.12), this can be done by multiplying the two factors Q_0 and R_0 in reverse order:

$$A_1 = R_0 Q_0 = Q_0^T A_0 Q_0. \tag{3.54}$$

Step 4: (Iteration) Repeat Steps 2 and 3 to obtain

$$A_i = R_{i-1} Q_{i-1} = Q_{i-1}^T A_{i-1} Q_{i-1}. \tag{3.55}$$

Step 5: (Stopping rule) Stop the iteration process when A_i is "diagonal enough," e.g., when the maximum off-diagonal elements of A_k is smaller than some error threshold ε in absolute value.

Step 6: (Calculation of the eigenvalues) The eigenvalues of A are approximately given by the diagonal elements of A_k, where k is the iteration order at which the iteration was stopped.

Step 7: (Calculation of the eigenvectors) The columns of the matrix

$$X = \prod_{i=0}^{k-1} Q_i \tag{3.56}$$

contain the eigenvectors corresponding to the eigenvalues found in Step 6.

The last step follows by noting that, if there is a similarity transform $X^{-1}AX = \Lambda$ transforming A to a diagonal matrix Λ, then the eigenvectors

of A are contained in the columns of X. In the present case, X is given by the product of the orthogonal matrices Q_i, $i = 1, 2, \dots, k$:

$$\underbrace{Q_{k-1}^T Q_{k-2}^T \cdots Q_1^T Q_0^T}_{X^T} A \underbrace{Q_0 Q_1 \cdots Q_{k-2} Q_{k-1}}_{X} = A_k \longrightarrow \Lambda \qquad (3.57)$$

Maple code 3.4.1: QR algorithm for symmetric matrices

```
use LinearAlgebra in
   QRdiag:=proc(A::Matrix(square,symmetric,float),tol::realcons)
      local i,imax,ndim,Q,R,X,Ap;
      imax:=10^4;
      ndim:=ColumnDimension(A);
      Ap:=A;
      X:=IdentityMatrix(ndim);
      for i to imax do
         Q,R:=QRDecomposition(Ap);
         Ap:=R.Q;
         X:=X.Q;
         if MaxOffDiagonal(Ap)<tol then
            return Diagonal(Ap),X;
         end if;
      end do;
      error "Not enough iterations!";
   end proc;
end use:
```

Maple code 3.4.2: `MaxOffDiagonal`

```
MaxOffDiagonal:=proc(A::Matrix(square,float))
   # Finds the largest (in magnitude) component of the matrix A
   local maxA,i,j,ndim;
   ndim:=LinearAlgebra[ColumnDimension](A);
   maxA:=abs(A[1,2]);
   for i to ndim-1 do
      for j from i+1 to ndim do
         if abs(A[i,j])>maxA then
            maxA:=abs(A[i,j]);
         end if;
      end do;
   end do;
   maxA;
end proc:
```

Chapter 4

Numerical integration

4.1. Introduction

The **indefinite integral** of a function $f(x)$ is denoted by

$$\int f(x)\ dx. \tag{4.1}$$

A solution $F(x)$ of this integral is called a **primitive** of $f(x)$:

$$F(x) = \int f(x)\ dx. \tag{4.2}$$

The primitive $F(x)$ is sometimes called the **anti-derivative** because it is such that $F'(x) = f(x)$.

If $F(x)$ is a primitive of $f(x)$, then so is $F(x)+c$, where c is any constant.

A **definite integral** is an integral performed over a given interval:

$$\int_a^b f(x)\ dx = F(b) - F(a). \tag{4.3}$$

Our goal in this chapter is to find ways to numerically evaluate a definite integral without knowing the primitive of the function to integrate.

Remark 4.1.1. The integration routine in Maple is called `int`. The syntax for evaluating the definite integral of $f(x)$ over the range $x \in [a, b]$ is

```
>int(f(x),x=a..b);
```

The syntax for indefinite integrals is `int(f(x),x)`. If Maple can find the solution of a indefinite integral, then the solution is shown as the output;

otherwise, Maple outputs the calling of the integral. If you use `Int` rather than `int`, Maple will only output the integral without evaluating it. `Int` is called the **unevaluated** form of the integral, whereas `int` is the **evaluated** form of the integral.

4.2. Riemann sums

You have seen in your calculus class that the definite integral

$$I = \int_a^b f(x)\, dx \tag{4.4}$$

can be interpreted in terms of an infinite **Riemann sum**:

$$\int_a^b f(x)\, dx = \lim_{n\to\infty} \sum_{k=0}^{n-1} f(x_k)(x_{k+1} - x_k). \tag{4.5}$$

The sum is illustrated in Figure 4.1. It involves n rectangles constructed with $n+1$ **midpoints** $x_0, x_1, \ldots, x_n$, with $x_0 = a$ as the first point and $x_n = b$ as the last one. The area of one rectangle is

$$A_k = \underbrace{f(x_k)}_{\text{height}} \underbrace{(x_{k+1} - x_k)}_{\text{base}}. \tag{4.6}$$

The integral, which corresponds to the area under the curve of $f(x)$, is recovered by increasing the number of rectangles:

$$\int_a^b f(x)\, dx = \lim_{n\to\infty} \sum_{k=0}^{n-1} A_k. \tag{4.7}$$

We assume throughout this chapter that the midpoints are **equally spaced** according to $x_k = a + kh$, for $k = 0, 1, \ldots, n-1$. The space between each point is

$$h = \frac{b-a}{n} \tag{4.8}$$

and $x_n = b$. With this choice of midpoints, the Riemann sum becomes

$$\int_a^b f(x)\, dx = \lim_{n\to\infty} h \sum_{k=0}^{n-1} f(x_k). \tag{4.9}$$

This Riemann sum is called the **left-point Riemann sum** because it involves rectangles whose height is given by the **left-most point** of each subintervals;

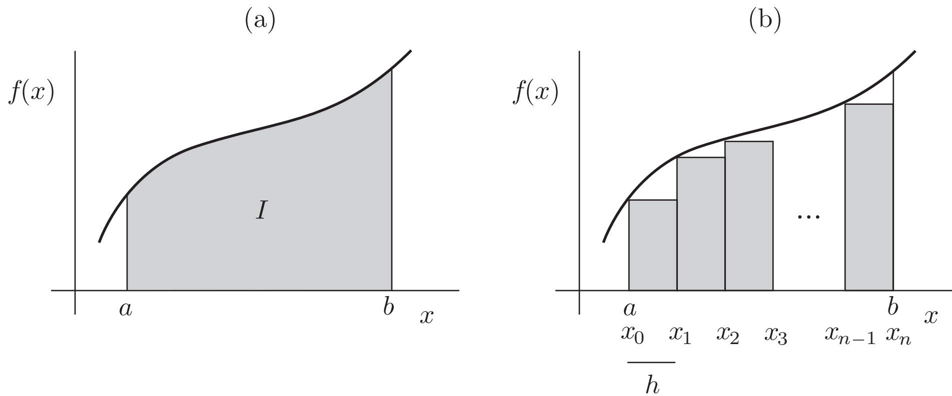

Figure 4.1: (a) Integral of $f(x)$ between a and b. (b) Riemann sum (left-point) approximating the integral of $f(x)$.

see Figure 4.1. In writing the Riemann sum, we can also choose the right-most point:

$$\int_a^b f(x)\ dx = \lim_{n\to\infty} \sum_{k=0}^{n-1} f(x_{k+1})(x_{k+1} - x_k) = \lim_{n\to\infty} h \sum_{k=0}^{n-1} f(x_{k+1}). \tag{4.10}$$

Alternatively, we can use the center-point of each subintervals:

$$\begin{aligned} \int_a^b f(x)\ dx &= \lim_{n\to\infty} \sum_{k=0}^{n-1} \frac{f(x_k) + f(x_{k+1})}{2}(x_{k+1} - x_k) \\ &= \lim_{n\to\infty} h \sum_{k=0}^{n-1} \frac{f(x_k) + f(x_{k+1})}{2}. \end{aligned} \tag{4.11}$$

Equations (4.10) and (4.11) are illustrated in Figure 4.2.

Riemann sums suggest an obvious way of evaluating integrals on a computer: truncate the infinite sum for a small but non-zero value of h, and evaluate the corresponding finite or **truncated** sum consisting of a large number of rectangles. In the case of the left-point Riemann sum, for example, this means that

$$\underbrace{\int_a^b f(x)\ dx}_{\text{Exact integral } I} \approx \underbrace{\sum_{k=0}^{n-1} f(x_k)(x_{k+1} - x_k)}_{\text{Truncated sum } I_n} = h \sum_{k=0}^{n-1} f(x_k). \tag{4.12}$$

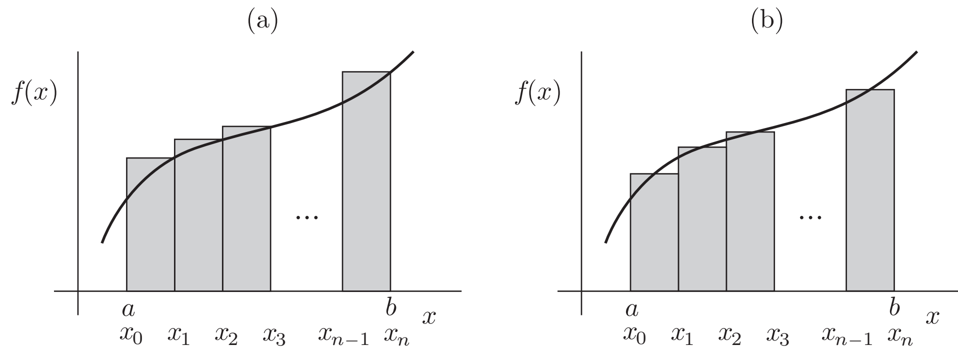

Figure 4.2: (a) Right-point Riemann sum. (b) Middle-point Riemann sum.

I_n is a **truncated approximation** involving $n+1$ midpoints or n rectangles. Similar formulae can be given for the right- and middle-point Riemann sums.

The **absolute truncation error** is

$$\delta I_n = |I - I_n|, \tag{4.13}$$

where I_n is any truncated Riemann sum (left-point, right-point or middle-point). As seen in Chapter 1, a good estimate of the truncation error is

$$\delta I_n \approx |I_n - I_{n-1}|. \tag{4.14}$$

For a finite number n of midpoints, the three versions of Riemann sums (left-point, right-point, middle-point) give different results in general when they are truncated. However, all three methods converge to the same value (the result of the integral) when using more and more points ($n \to \infty$).

4.3. Trapeze integration rule

In the truncated Riemann sum, the function $f(x)$ is approximated as a constant over each subinterval of integration; see Figure 4.1. This leads to a good approximation of the integral of $f(x)$ over $[a, b]$ if h, the spacing of the midpoints, is very small.

A better approximation of the integral can be obtained by constructing a better approximation of $f(x)$ over each subinterval of integration. For example, we can approximate $f(x)$ between two midpoints x_k and x_{k+1} by the line passing through the points $(x_k, f(x_k))$ and $(x_{k+1}, f(x_{k+1}))$; see Figure 4.3. This approximation is the basis for the **trapeze rule of integration**.

Maple code 4.2.1: Riemann integration (long version)

```
riemannint:=proc(f::procedure,a::realcons,b::realcons,n::integer)
   local i,h,sumval,x;
   h:=(b-a)/n;
   sumval:=0;
   for i from 0 to n-1 do
      x:=a+i*h;
      sumval:=sumval+f(x)*h;
   end do;
   sumval;
end proc:
```

Maple code 4.2.2: Riemann integration (short version)

```
riemannint:=proc(f::procedure,a::realcons,b::realcons,n::integer)
   local h,resint;
   h:=(b-a)/n;
   resint:=h*sum(f(a+i*h),i=0..n-1);
   return resint;
end proc:
```

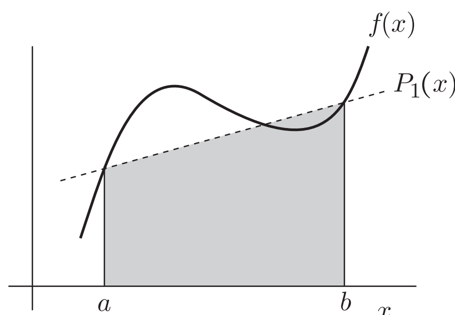

Figure 4.3: Trapeze integration.

Construction of the trapeze rule

Given $f(x)$ and two points a and b, we build the trapeze rule by first approximating $f(x)$ between a and b by the following polynomial of degree 1:

$$P_1(x) = f(a)\frac{x-b}{a-b} + f(b)\frac{x-a}{b-a}. \tag{4.15}$$

This polynomial represents the equation of a line passing through $(a, f(a))$ and $(b, f(b))$; see Figure 4.3. Because of this property, we say that $P_1(x)$ **interpolates** $f(x)$ at a and b.

Next, we approximate the integral of $f(x)$ between a and b as the integral of $P_1(x)$ between these two bounds. That is,

$$\begin{aligned}\int_a^b f(x)\; dx &\approx \int_a^b P_1(x)\; dx \\ &= \int_a^b \left[f(a)\frac{x-b}{a-b} + f(b)\frac{x-a}{b-a}\right]\; dx \\ &= \frac{b-a}{2}[f(a)+f(b)]. \end{aligned} \tag{4.16}$$

The resulting approximation for the integral is thus

$$\int_a^b f(x)\; dx \approx \frac{h}{2}[f(a)+f(b)]. \tag{4.17}$$

This approximation is called the **trapeze integration rule** because the area below the graph of $f(x)$ is approximated as the area of the trapeze built from the linear interpolation; see Figure 4.3. Note that the trapeze approximation has the same form as the center-point Riemann sum.

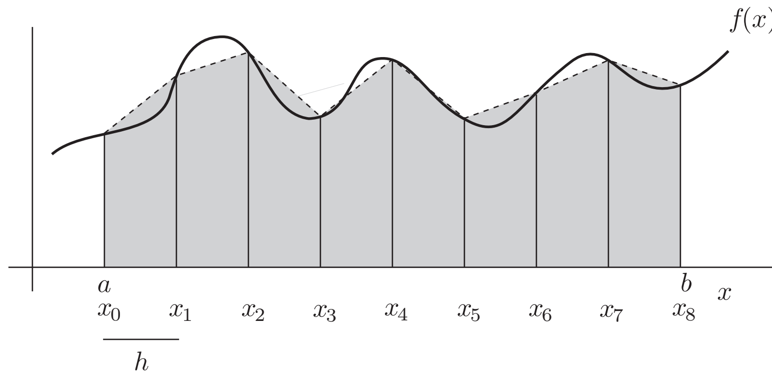

Figure 4.4: Composite trapeze integration.

Composite trapeze rule

The composite trapeze integration is built by decomposing the complete integral

$$I = \int_a^b f(x)\ dx \tag{4.18}$$

into n subintegrals

$$I = \int_{x_0}^{x_1} f(x)\ dx + \int_{x_1}^{x_2} f(x)\ dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)\ dx, \tag{4.19}$$

and by applying the trapeze integration rule to each of the subintegrals. This yields

$$I \approx \frac{h}{2}[f(x_0)+f(x_1)] + \frac{h}{2}[f(x_1)+f(x_2)] + \cdots + \frac{h}{2}[f(x_{n-1})+f(x_n)]. \tag{4.20}$$

In this expression, the nodes repeat themselves twice except for $x_0 = a$ and $x_n = b$ which appear only once. As a result,

$$\int_a^b f(x)\ dx \approx \frac{h}{2}\left[f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_i)\right]. \tag{4.21}$$

Estimate of the error

To estimate the error associated with the trapeze approximation, we have to use the following result:

$$f(x) = P_1(x) + R_1(x), \tag{4.22}$$

Maple code 4.3.1: Trapeze integration (long version)

```
trapezeint:=proc(f::procedure,a::realcons,b::realcons,n::integer)
   local i,h,sumval,x;
   h:=(b-a)/n;
   for i from 1 to n-1 do
      x:=a+i*h;
      sumval:=sumval+2*f(x);
   end do;
   sumval:=h*(sumval+f(a)+f(b))/2;
   return sumval;
end proc:
```

Maple code 4.3.2: Trapeze integration (short version)

```
trapezeint:=proc(f::procedure,a::realcons,b::realcons,n::integer)
   local h,resint;
   h:=(b-a)/n;
   resint:=h/2*(f(a)+f(b)+2*sum(f(a+i*h),i=1..n-1));
   return resint;
end proc:
```

where

$$R_1(x) = \frac{f''(\xi)}{2}(x-a)(x-b), \quad \xi \in (a,b). \tag{4.23}$$

$P_1(x)$ is our linear approximation for $f(x)$ between a and b, and $R_1(x)$ is the error or **remainder** associated with that approximation. As expected, $R_1(x) = 0$ at $x = a$ and $x = b$ because $P_1(a) = f(a)$ and $P_1(b) = f(b)$. In between a and b, $R_1(x)$ is in general different than zero, unless $f(x)$ is a linear function, in which case $f''(x) = 0$.

Using the expression of the remainder $R_1(x)$, we evaluate the error of the trapeze rule simply by integrating $R_1(x)$ over $[a,b]$:

$$\underbrace{\int_a^b f(x)\ dx}_{\text{exact integral}} = \underbrace{\int_a^b P_1(x)\ dx}_{\text{trapeze approximation}} + \underbrace{\int_a^b R_1(x)\ dx}_{\text{error } \delta I}. \tag{4.24}$$

We are not interested at this point in calculating the integral of $R_1(x)$ exactly. What is essential is to know how the error δI varies with $h = b - a$ as $h \to 0$. In our case, $R_1(x)$ is a polynomial of degree 2, so that

$$\delta I = \int_a^b R_1(x)\ dx \sim h^3 \tag{4.25}$$

as $h \to 0$.

With this basic result, we estimate the error associated with the composite application of the trapeze approximation as follows. Each trapeze in the composite integral gives rise to an absolute error proportional to h^3 when h is small. There are n such trapezes, and $h = (b-a)/n \sim n^{-1}$, so that

$$\delta I_{\text{tot}} \approx nh^3 \sim n\frac{1}{n^3} = n^{-2}. \tag{4.26}$$

This estimate is valid for n large, and predicts what should be expected, namely that the error decreases by increasing the number of nodes.

4.4. Simpson's integration rule

Simpson's integration rule improves on the trapeze integration rule by considering a quadratic approximation of $f(x)$ in each subinterval of integration instead of a linear approximation; see Figure 4.5.

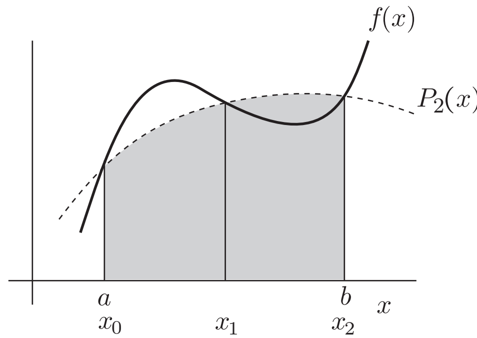

Figure 4.5: Simpson's integration.

Construction of Simpson's rule

To construct a parabola interpolating $f(x)$, we need three equally-spaced points or **nodes**, denoted by

$$x_0 = a, \quad x_1 = \frac{b+a}{2}, \quad x_2 = b. \tag{4.27}$$

The equation of the parabola, in terms of these three points, is

$$\begin{aligned} P_2(x) = & \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) \\ & + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2). \end{aligned} \tag{4.28}$$

This is a polynomial of degree 2 (a parabola), which interpolates $f(x)$ at the three nodes; see Figure 4.5.

Following the trapeze case, we use $P_2(x)$ as our approximation of $f(x)$ over the integration interval, and integrate this approximation:

$$\int_a^b f(x)\, dx \approx \int_a^b P_2(x)\, dx. \tag{4.29}$$

Peforming the integral of $P_2(x)$ yields **Simpson's rule**:

$$\int_a^b f(x)\, dx \approx \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)]. \tag{4.30}$$

Composite Simpson's rule

To establish the formula for the composite application of Simpson's rule, we proceed similarly as for the composite trapeze rule by considering $n+1$ equally spaced nodes. However, Simpson's rule requires three nodes, so that the number of integration subintervals that we must consider is not n but half that number, that is, $m = n/2$. Thus, for a start, we need n to be even, otherwise there will be a node left alone in trying to apply the composite rule.

The m integration subintervals divide the complete integral as following:

$$\int_a^b f(x)\ dx = \int_{x_0}^{x_2} f(x)\ dx + \int_{x_2}^{x_4} f(x)\ dx + \cdots + \int_{x_{n-2}}^{x_n} f(x)\ dx. \tag{4.31}$$

Note the limits of the subintegrals: they never involve nodes x_k having an odd index k.

Next, we apply Simpson's rule to each subintervals to obtain

$$I \approx \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)] + \frac{h}{3}[f(x_2) + 4f(x_3) + f(x_4)] + \\ + \frac{h}{3}[f(x_4) + 4f(x_5) + f(x_6)] + \cdots . \tag{4.32}$$

As for the trapeze case, some boundary points of the subintervals repeat themselves in this sum. Grouping them together yields

$$\int_a^b f(x)\ dx \approx \frac{h}{3}\left[f(a) + f(b) + 2\underbrace{\sum_{k=1}^{m-1} f(x_{2k})}_{\text{even indices}} + 4\underbrace{\sum_{k=1}^{m} f(x_{2k-1})}_{\text{odd indices}} \right], \tag{4.33}$$

where, again, $m = n/2$.

Estimation of the error

The absolute error δI associated with Simpson's rule is estimated just like in the trapeze case. Here we have

$$f(x) = P_2(x) + R_2(x) \tag{4.34}$$

with the remainder given by

$$R_2(x) = \frac{f'''(\xi)}{6}(x - x_0)(x - x_1)(x - x_2), \quad \xi \in (x_0, x_2). \tag{4.35}$$

Maple code 4.4.1: Simpson's integration (long version)

```
simpsonint:=proc(f::procedure,a::realcons,b::realcons,n::integer)
   local h,m,sumval,i,x;
   h:=(b-a)/n;
   m:=n/2;
   sumval:=0.;
   if even(n) then
      for i from 1 to m-1 do
         x:=a+2*i*h;
         sumval:=sumval+2*f(x);
      end do;
      for i from 1 to m do
         x:=a+(2*i-1)*h;
         sumval:=sumval+4*f(x);
      end do;
      sumval:=h*(sumval+f(a)+f(b))/3;
   else
      error "Simpson's rule requires n to be even.";
   end if;
   sumval;
end proc:
```

Maple code 4.4.2: Simpson's integration (short version)

```
simpsonint:=proc(f::procedure,a::realcons,b::realcons,n::integer)
   local h,resint,evensum,oddsum;
   h:=(b-a)/n;
   if even(n) then
      evensum:=sum(f(a+2*j*h),j=1..n/2-1);
      oddsum:=sum(f(a+(2*j-1)*h),j=1..n/2);
      resint:=h/3*(f(a)+f(b)+2*evensum+4*oddsum);
   else
      error "Simpson's rule requires n to be even.";
   end if;
   resint;
end proc:
```

Hence,

$$\underbrace{\int_a^b f(x)\,dx}_{\text{exact integral}} = \underbrace{\int_a^b P_2(x)\,dx}_{\text{Simpson approximation}} + \underbrace{\int_a^b R_2(x)\,dx}_{\text{error } \delta I}. \qquad (4.36)$$

$R_2(x)$ is now a degree 3 polynomial, so that

$$\delta I = \int_a^b R_2(x)\,dx \sim h^4 \qquad (4.37)$$

as $h \to 0$. This is the estimate of the error for one application of Simpson's rule.

In the composite Simpson's rule, each subinterval gives rise to an error of order h^4, and there are $m = n/2$ such subintervals, so that

$$\delta I_{\text{tot}} \approx \frac{n}{2} h^4 \sim n \frac{1}{n^4} = n^{-3}. \qquad (4.38)$$

This is a good estimate of the total error when n becomes large.

4.5. *Newton-Cotes integration

The trapeze integration rule and Simpson's integration rule are two examples of what is called a **Newton-Cotes integration rule** built from a **Lagrange interpolating polynomial**. The trapeze rule is a first-order Newton-Cotes

integration rule built from a Lagrange interpolating polynomial of degree 1 (a line). Simpson's rule is a second-order Newton-Cotes integration rule built from a Lagrange interpolating polynomial of degree 2 (a parabola). Continuing on this idea, we construct an n-order Newton-Cotes integration rule by approximating $f(x)$ with a polynomial of degree n, which interpolates $f(x)$ at $n+1$ points. These interpolating polynomial are defined next.

Lagrange interpolating polynomials

Suppose that we have $(n+1)$ equally-spaced points or nodes $x_0, x_1, \ldots, x_n$ where, as before, $x_0 = a$ and $x_n = b$. The **Lagrange polynomial** defined with these $n+1$ points has the form

$$P_n(x) = \sum_{k=0}^{n} L_{n,k}(x) f(x_k), \tag{4.39}$$

where

$$L_{n,k}(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}. \tag{4.40}$$

The product in the numerator involves all the midpoints $x_0, x_1, \ldots, x_n$ except x_k. The product in the denominator is constructed similarly except that x is replaced by the missing midpoint x_k of the numerator. One telling way of writing $L_{n,k}(x)$ is

$$L_{n,k}(x) = \frac{\displaystyle\prod_{i=0, i\neq k}^{n} (x - x_i)}{\displaystyle\prod_{i=0, i\neq k}^{n} (x_k - x_i)}. \tag{4.41}$$

The Lagrange polynomial $P_n(x)$ has two special properties:

- $P_n(x)$ is a polynomial of degree n;
- $P_n(x)$ passes through the graph of $f(x)$ at all the midpoints. Mathematically, this simply means that

$$f(x_k) = P_n(x_k), \qquad k = 0, 1, \ldots, n. \tag{4.42}$$

To prove this property, note that, by construction of $L_{n,k}(x)$,

$$L_{n,k}(x_i) = \begin{cases} 0 & k \neq i \\ 1 & k = i, \end{cases} \tag{4.43}$$

so that

$$P_n(x_i) = \sum_{k=0}^{n} L_{n,k}(x) f(x_k) = L_{n,i}(x_i) f(x_i) = f(x_i). \tag{4.44}$$

The second property is what makes $P_n(x)$ an **interpolating** polynomial: it matches $f(x)$ at all the midpoints or nodes $x_0, x_1, \ldots, x_n$.

As we have noticed already, $P_n(x)$ may be different from $f(x)$ away from the nodes, so that, in general, we have

$$f(x) = P_n(x) + R_n(x) \tag{4.45}$$

with $R_n(x) \neq 0$ away from the nodes and $R_n(x) = 0$ at the nodes. $R_n(x)$ is what we called the **remainder** of the interpolation.

If $f(x)$ is a polynomial of degree n, then $P_n(x) = f(x)$ not only at the nodes but for all $x \in [a, b]$. This is because a Lagrange polynomial defined with a given set of nodes is unique.

Example 4.5.1. The Lagrange interpolating polynomial built from the two nodes a and b is

$$P_1(x) = f(a)\frac{x-b}{a-b} + f(b)\frac{x-a}{b-a}. \tag{4.46}$$

This is the equation of the line that we used to construct the trapeze integration rule; see Eqn. (4.15).

Example 4.5.2. The Lagrange interpolating polynomial of degree 2 built from the three nodes x_0, x_1 and x_2 has for expression

$$\begin{aligned} P_2(x) = & \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) \\ & + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2). \end{aligned} \tag{4.47}$$

This is the equation of the parabola that we used for constructing Simpson's integration rule; see Eqn. (4.28).

n-point Newton-Cotes integration

Let $P_n(x)$ be a Lagrange polynomial interpolating $f(x)$ at the nodes $\{x_0, x_1, \ldots, x_n\}$, that is,

$$f(x) = P_n(x) + R_n(x) \tag{4.48}$$

with $R_n(x_k) = 0$ at each node x_k. Using this interpolation, we can rewrite the integral of $f(x)$ over $[a, b]$ as

$$\begin{aligned}\int_a^b f(x)\ dx &= \int_a^b P_n(x)\ dx + \int_a^b R_n(x)\ dx \\ &= \int_a^b \sum_{k=0}^{n} f(x_k) L_{n,k}(x)\ dx + \int_a^b R_n(x)\ dx \\ &= \sum_{k=0}^{n} f(x_k) \int_a^b L_{n,k}(x)\ dx + \int_a^b R_n(x)\ dx \\ &= \sum_{k=0}^{n} f(x_k) w_k + \int_a^b R_n(x)\ dx, \end{aligned} \tag{4.49}$$

where

$$w_k = \int_a^b L_{n,k}(x)\ dx. \tag{4.50}$$

Neglecting the remainder or error term involving $R_n(x)$, we then obtain

$$I = \int_a^b f(x)\ dx \approx \sum_{k=0}^{n} f(x_k) w_k. \tag{4.51}$$

This approximation of the integral I is known as the **n-point Newton-Cotes integration rule**. Note its similarity with the Riemann sum. In the Riemann case, the **weight** is given by $w_k = (x_{k+1} - x_k)$, whereas for the Newton-Cotes case w_k is given by Eqn. (4.50).

Like all the integration approximations seen so far, the Newton-Cotes approximation becomes exact in the limit where the number of nodes becomes infinite (assuming that $f(x)$ is smooth enough). This is because $R_n(x) \to 0$ as $n \to \infty$ uniformly for all $x \in [a, b]$, so that

$$\lim_{n\to\infty} \int_a^b R_n(x)\ dx = 0. \tag{4.52}$$

For a finite number of nodes,

$$\int_a^b R_n(x)\ dx \neq 0 \tag{4.53}$$

in general, so there is a non-zero error associated with the Newton-Cotes approximation.

4.6. *Adaptive integration

See Section 4.5 of R.L. Burden, J.D. Faires, Numerical Analysis, Prindle, Weber & Schmidt, 1985.

Maple code 4.6.1: Adaptive integration

```
adaptint:=proc(f,l,r,tol)
   local h,m,area,areatot,nextareatot,err,arealeft,arearight;
   h:=(r-l);
   m:=(r+l)/2;
   area:=0.;
   areatot:=trapezearea(f,l,r);
   nextareatot:=trapezearea(f,l,m)+trapezeare(f,m,r);
   err:=abs(areatot-nextareatot);
   if err<tol then
      return areatot;
   else
      arealeft:=adaptint(f,l,m,tol/2);
      arearight:=adaptint(f,m,r,tol/2);
      area:=area+arealeft+arearight;
   end if;
   area;
end proc:
```

Maple code 4.6.2: Trapeze area formula

```
trapezearea:=proc(f,l,r)
   local h,area;
   h:=(r-l);
   area:=h*(f(l)+f(r))/2;
   return area;
end proc:
```

Maple programs

2.2.1 Fixed-point iteration method 16
2.3.1 Bisection method 18
2.4.1 Newton-Raphson method 21
2.7.1 Newton-Raphson method for 3D vectors 27
3.2.1 Power method (basic version) 35
3.2.2 Power method (full version) 36
3.2.3 `maxind` 37
3.3.1 Wielandt deflation algorithm (eigenvalues only) 43
3.3.2 Wielandt deflation algorithm (eigenvalues and eigenvectors) . 44
3.4.1 QR algorithm for symmetric matrices 46
3.4.2 `MaxOffDiagonal` 47
4.2.1 Riemann integration (long version) 53
4.2.2 Riemann integration (short version) 53
4.3.1 Trapeze integration (long version) 56
4.3.2 Trapeze integration (short version) 56
4.4.1 Simpson's integration (long version) 60
4.4.2 Simpson's integration (short version) 61
4.6.1 Adaptive integration 65
4.6.2 Trapeze area formula 65

Index

absolute error, 6
algebraic equation, 13
algebraic functions, 13
anti-derivative, 49
approximation, 6

characteristic equation, 29
characteristic polynomial, 29
complete basis, 30
complex roots, 23

decomposition, 31
definite integral, 49
Deflation, 34
diagonal matrix, 31
dominant eigenvalue, 29, 32

eigenvalue, 29
eigenvector, 29

fixed-point, 14
fixed-point iteration method, 13

ill-conditioned problem, 10
ill-defined problem, 11
ill-posed, 11
indefinite integral, 49
initial value, 13
Integers, 5
Irrational numbers, 5
iterates, 13
iteration function, 13

Jacobian matrix, 24

Lagrange interpolating polynomial, 61
Lagrange polynomial, 62
left-point Riemann sum, 50
linearly independent, 30

map, 13
midpoints, 50
multiplicity, 29

Newton-Cotes integration rule, 61
Newton-Raphson method, 19
nodes, 58
non-associative, 8
norm, 25
normalisation step, 34

orthogonal, 31

power method, 31
primitive, 49
projections, 30

Rational numbers, 5
relative error, 6
Riemann sum, 50
root, 13
rounding error, 5

secant method, 22
seed, 13
similarity transform, 30
Simpson's integration rule, 57
Simpson's rule, 58

spectral radius, 26
stable iteration, 10
stopping rule, 15
symmetric, 30

threshold, 15
total error, 9
trapeze integration rule, 54
trapeze rule of integration, 52
truncated decimal representations, 5
truncation error, 9, 15

unstable iteration, 10
upper triangular, 31

zero, 13